

sinclair

ZX Spectrum Next

Written and Illustrated by
Phoebus R. Dokos, BSc (Hons)

With extracts from:
ZX Spectrum – 3 User manual
by Ivor Spital, Cliff Lawson and Rupert Goodwins
ZX Spectrum BASIC programming manual
by Steven Vickers and Robin Bradbeer

User Manual

Edited by:

Mike Cadwallader, Uwe Geiken
Darren Grayson, Matt Langley
David Sadler, Paulo Silva
Julian Smith and Steve Smith

With invaluable contributions by:

Alvin Albrecht, Garry Lancaster, Simon N Goodwin,
Simon Brattel and Kev Brady

Copyright © 2020-2024 Phoebus Dokos / SpecNext Ltd – London, United Kingdom

This work is licensed under a CC BY-NC-SA 4.0 International License,
<http://creativecommons.org/licenses/by-nc-sa/4.0/>

Cover Illustration: Jonathan M Betts (www.artstation.com/jonahambettsart)
Cover Layout: Phoebus R Dokos (www.dokos-gr.net)

Used as follows:

ZK Spectra Next History
Copyright © 2017-2024 by Phoebus R Dokos

THIRD EDITION

ISBN: 978-1-5272-5496-1

To Georgia

Copyrights / Trademarks

Sinclair and ZX Spectrum are copyright © Amstrad/Sky plc and are used under license

Spectrum Next and System/Next are copyright © SpecNext Ltd

The NextCore is © Alvin Albrecht

+3e, ResiDOS, IDEDOS, NextZXOS and NextBASIC are copyright © Garry Lancaster

TBBLUE is © Victor Trucco and Fabio Belavenuto

Zeus is © Neil Mottershead and Simon Brattel

NextPi, NextPi2, SPUi and QE are © D. Rimmer

ZX-UNO is © The ZX-UNO Team (Superfo, Avillena, McLeod, Quest, Harko)

divMMC is © Mario Prato

CP/M is © Intel Inc.

esxDOS is © Miguel Guerreiro / Papaya Design

The ZX80/ZX81 emulators are © Paul Farrow

Gosh Wonderful and Looking Glass are © Geoff Wearmouth

rdtp and NxtTel are © Robin Verhagen-Guest

NextGuide, ED and Odin are © Matt Davies

SpectraMan and ZIP/Next are © Simon N Goodwin

ZXDB-di and GetIt are © David Saphier

vDrive® is © Charlie Ingley

ULApus is ™ ZX Design and Media

All other names and trademarks used herein are ®/™ of their respective authors/owners

Chapter 1 – Basic Programming Concepts

Introduction

If you read through the *Quick Start Guide*, included with your new ZX Spectrum Next, you've already had a brief introduction of the screen, keys, editing and *NextBASIC* in general which means you're ready to start programming your computer! If not, you can either go along and you'll figure things you've missed along the way – or – go back and have a quick read of the *A (Next)BASIC Primer* section! Either way, you need to reset your ZX Spectrum Next, go to the *Startup Menu* and select *NextBASIC*. Press **ENTER** and we're ready to start!

PRINT, LET, programs and line numbers

Type in the following two lines:

```
20 PRINT a
10 LET a=10
```

so that the screen looks like this:



Fig. 1 – Entering program lines in *NextBASIC*

First of all congratulations! You just wrote a *computer program* which stores a number in the computer's memory, later recalls it and displays it. Let's see for a moment exactly how you've done that:

- Since these lines began with numbers (as you already know from the *Quick Start Guide*), they were not obeyed immediately but stored as *program lines*. You will also have noticed here that the respective line numbers govern the order of the lines within the program; the lower the number the earlier (higher in the list) it appears. This matters most when the program is run, but it also governs the order of the lines that you see on the screen now.
- By using the command **LET** you've instructed the computer to await an *assignment* – the assignment itself is indicated by the = (equals sign). *Assignment* is the pairing of the numeric value **10** to a variable named **a**.

Let's enhance our program a bit more. Type:

```
15 b=15
```

and press **ENTER**. Line 15 gets inserted between lines 10 and 20 and the screen is reformatted. If the lines' numbers had only an interval of 1; if for example they had been numbered 1 and 2 instead of 10 and 20 it would have been impossible to insert another line in-between. Line numbers must be whole numbers between 1 and 9999, and that is why,

when first typing-in a program, it is good practice to leave large enough intervals in-between the numbers.

You'll also notice, that for line 15 there's no LET keyword although the = remains. That is because LET is optional and it's implied from the assignment alone. Functionally therefore, lines 10 and 15 are identical. For this chapter, we will keep using LET so you can see the assignments clearly but further on, we will skip them altogether as they make for much more readable code.

Note here that we will do the same with the printed representation of *Syntax Highlighting* as it requires a contrast with the background which a printed manual doesn't provide.

Variables and Arrays

Before we continue further, let's take a pause and discuss what the letters a and b in the examples above are called. We call these *variables* because they represent locations in the computer's memory where we can temporarily store information to be recalled and used at any time a program is being executed. There are two types of variables by usage: *Global* and *Local*. Global variables apply to an entire NextBASIC program and these are the ones we will talk about here. Local variables, apply only to subprogram areas we call *procedures* and *functions* and they will be discussed in the respective chapters.

NextBASIC can store two types of information in memory: *numbers* and *text*. Numbers are further separated into *floating point* and *integers*. Text variables are called *strings* and they will be discussed in Chapter 7. Furthermore, NextBASIC can group together variables of the same type and refer to them collectively. These groupings are called *arrays*.

There are some restrictions in the naming and quantity of available variables and arrays as you can see in the following table according to their type. It is advisable to make use of integer variables over their regular numeric counterparts despite their restrictions¹ at least where speed of execution is concerned.

	Integer variables	Numeric variables	String Variables
Qty	Fixed 26	Limited only by memory	Limited only by memory
Naming	Single character prefixed by the % symbol	Combination of characters and numbers	Combination of characters and numbers suffixed by the \$ symbol
Arrays	Fixed 26 with maximum 64 elements (0..63) Expandable size and dimensions (by reducing the number of available arrays)	Limited only by memory (Indices are 1-based)	Limited only by memory (Indices are 1-based)

Table 1 – Types of NextBASIC variables

Assignments

We saw earlier that using LET with a variable together with the symbol = and some value is called an assignment. What actually happens is that LET instructs NextBASIC to move a value (numeric or character) into a location in the computer's memory which we can later identify and recall by an easy-to-use name (See Table 1 above). Unlike previous versions of Sinclair BASICs that required the LET command and only allowed a single LET command per assignment, NextBASIC allows the omission of the LET keyword altogether (as the assignment operator = implies its use anyway) while, at the same time, multiple assignments of variables per LET command. In other words, the full form of LET is:

[LET] variable1 [{variable2...variablen}] = value1 [{value2...valuen}]

Moreover, assignments allow multiple destinations and a mix of types of value and variables. Some examples of the above are:

```
LET x,y=10,20
```

¹ Integer variables in NextBASIC are 16-bit (unsigned or signed). That means that they accept values from 0 to 65535 (or from -32768 to 32767).


```
X, Y = 10, 20
```

which are equivalent, or in a more descriptive manner:

```
LET numberA, numberB, stringA, stringB =  
1, 2, "hello", "goodbye"
```

One, extremely handy functionality of assignments is that in the case of assignment of multiple variables, if there are fewer values after the = operator than the variables before it, then all the remaining variables get initialised to that specific value. The following line:

```
a, b, c, d, e, f = 0
```

will create variables a, b, c, d, e and f and set them all to 0. Similarly the following line:

```
a, b, c, d, e, f = 10, 20, 30, 1
```

will assign 10 to variable a, 20 to variable b, 30 to variable c and 1 to variables d, e and f. In the example above we could remove altogether line 10 and instead enter

```
15 a, b = 10, 15
```

which is functionally equivalent to both lines 10 and 15!

Finally, assignments can be made cumulative with the use of special combination operators as seen in the table below:

Assignment Operator	Numeric Variables	String Variables
+=	Increases variable by the value assigned	Concatenates string variable with the string assigned
-=	Decreases variable by the value assigned	– Not Applicable –
*=	Multiplies variable by the value assigned	Replicates the string variable as many times as the numeric value assigned
/=	Divides variable by the value assigned	– Not Applicable –
^=	Raises variable to the power assigned	– Not Applicable –
&=	ANDs variable with the value assigned	– Not Applicable –
=	ORs variable with the value assigned	– Not Applicable –
^ =	XORs variable with the value assigned	– Not Applicable –
<<=	Shifts left the variable as many positions as the value assigned.	– Not Applicable –
>>=	Shifts right the variable as many positions as the value assigned.	– Not Applicable –
MOD=	Performs a MODulo operation on the variable with the value assigned.	– Not Applicable –

Table 2 – Accumulation assignments

This allows us to be a bit more terse when writing a program by reducing the amount of text we have to type, making for some more readable code. Consider the following example:

```
10 LET a = 10  
15 LET b = 15  
20 LET c = a  
30 LET a = a + b  
40 LET b = b + c
```

Using the information we've just learned, we can rewrite it to use multiple assignment statements as well as cumulative assignment operators like so:


```
15 a, b = 10, 15
20 a, b += b, a
```

It's obvious from the example above that after skipping both the **LET** keyword and the long form of assignment, our program suddenly became more readable and much easier to write!

Labels

Apart from the line numbers (which we will see how to refer – and jump to – in the following sections) sometimes we need a way to identify and/or jump to a selected *NextBASIC* statement, maybe even within a multi-statement line. For this reason, *NextBASIC* provides us with a facility called *labels*. Labels are identified by the **at** (**@**) symbol prefix and a name following the naming guidelines for a *procedure* (See Chapter 4) and they are defined within a program starting with either the line number or the colon statement separator (:) if they're not defined at the start of a line. Their definition can appear anywhere within a program:

```
20 @oneLabel: PRINT a+b
30 PRINT a+b: @anotherLabel
```

Labels can be used in lieu of line numbers with the following keywords: **GO TO**, **BANK...GO TO**, **GOSUB**, **BANK...GOSUB**, **LIST**, **BANK...LIST**, **SAVE...LINE²** and **EXIT**. See relevant section for each keyword's proper syntax. **BANK** commands are all discussed in length in Chapter 23 – *The Memory*.

Using LIST, RUN and cursors to edit and run programs

Going back to our program, you will need to change line 20 to:

```
20 PRINT a+b
```

You could type out the replacement in full, but it is easier to move the cursor (using the cursor keys) to just after the **a**, and then type:

+ b (without **ENTER**)

The line at the bottom should now read:

```
20 PRINT a+b
```

Press **ENTER** and it will replace the old line 20, so that the screen looks like this:



Fig. 2 – Editing a program

² In the case of **SAVE...LINE@label**, the saved program will substart from the beginning of the line that contains the label even if it's not the first statement in the line.

Run this program using **RUN** and **ENTER** and the sum will be displayed (25). Run the program again and then type:

```
PRINT a, b
```

The variables are still there, even though the program has finished. If you enter a line by mistake, say:

```
12 b=0
```

it will go up into the program and you will realise your mistake. To delete this unnecessary line, type:

```
12 (with ENTER of course)
```

Line 12 will disappear, and the cursor will appear where line 12 used to be.

Now type:

```
30 (and ENTER)
```

This time, the program cursor will appear after the end of the program (having tried to find line 30 and failed). If you enter any line number that does not exist, *NextBASIC's Editor* will place the cursor where it thinks the line would have been if it existed. This can be a useful way of moving around large programs, but beware – it can be very dangerous because if the line really did exist before you entered the number, it wouldn't exist afterwards (refer to the line 12 example above)!

To list a program on screen, type

```
LIST
```

and press **ENTER**. You may wish to list a program from a certain point onwards. This can be achieved by typing an appropriate line number after the **LIST** command. Try

```
LIST 15 (and ENTER)
```

to see this in action. If, at some point, you find you haven't left enough space between line numbers then you may use the edit menu to renumber a program. To do this, press the **EDIT** key then select the *Renumber* option from the menu that appears; this sets the gap between each line number to 10. Try this out and see how the line numbers change.

REM, NEW, INPUT and GO TO

The command **NEW** erases any old programs and variables in the computer and starts the machine anew. Try it now; type:

```
NEW
```

and press **ENTER**. You'll see the *Welcome Screen* and then the *Startup menu*. With the menu on screen, select again the *NextBASIC* option.

Carefully type in this program, which changes Fahrenheit temperatures to Celsius:

```
10 REM Temperature Conversion
20 PRINT "deg F", "deg C"
30 PRINT
40 @inpF: INPUT "Enter deg F",
   F
50 PRINT F, (F-32)*5/9
60 GO TO @inpF
```


Now run it. You will see the headings printed on the screen by line 20, but what happened to line 10? Apparently the computer has completely ignored it changing its colour to red. Indeed, **REM** in line 30 stands for **REMark** and is there solely to remind you of what the program does. A **REM** command consists of **REM** or the semicolon symbol (**;**) followed by anything you like, and the computer will ignore it right up to the end of the line.

REM is not really part of a *NextBASIC* program, it just adds remarks to it for improved readability and documentation and gets totally ignored by *NextBASIC*. For example:

```
10 REM this is a remark
20 ; This is also a remark
```

are functionally equivalent, as are:

```
10 PRINT 10: REM Remark
20 PRINT 20: ; Remark
```

Note that the colon (**:**) cannot be omitted, like:

```
10 PRINT 10; Remark
```

as then **Remark** forms part of the **PRINT** statement. A colon must **ALWAYS** be used to separate statements on the same line.

Using STOP, BREAK and CONTINUE

By now, the computer has got to the **INPUT** command on line 40 and is waiting for you to type in a value for the variable **F** – you can tell this because at the bottom of the screen is a flashing cursor. Enter a number; remember to press **ENTER** afterwards! Now the computer has displayed the result and is waiting for another number. This is because of line 60, **GO TO @inpF**, which means exactly what it says. Instead of running out of program and stopping, the computer jumps back to line 40 where the label **@inpF** is located and starts again. So, enter another temperature. After a few more of these you might be wondering if the machine will ever get bored with this, it won't. Next time it asks for another number, enter the word **stop**. The computer will stay in the line and the cursor will change shape indicating a non acceptable entry. You have there the choice of hitting **BREAK** in which case you receive a report **H STOP in INPUT, 40:2**, which tells you why it stopped, and where (in the second statement of line 40, first being the label **@inpF**).

If you want to continue the program type:

CONTINUE

and the computer will continue with the **INPUT** line.

There's a synonym of **CONTINUE** which is really there for convenience and it's **CONT**. Try it in lieu of **CONTINUE** above; it will work in the same way.

Replace line 60 by **GO TO 21** – it will make no perceptible difference to the running of the program. If the line number in a **GO TO** command refers to a non-existing line, then the jump is to the next line after the given number.

This, however, is **NOT** the case when using **GO TO** to jump to a label as the latter **MUST** exist otherwise an error will be produced. The same allowance for line numbers is true as well for **RUN**; in fact **RUN** on its own actually means **RUN 0**.

Now type in numbers until the screen starts getting full. When it is full, the computer will move the whole of the top half of the screen up one line to make room, losing the heading off the top. This is called scrolling.

When you are tired of this, stop the program as shown above and get the listing by pressing **ENTER**.

Look at the **PRINT** statement on line 50. The punctuation or *print modifier* in this — the comma — is very important and you should remember that it follows much more definite rules than the punctuation in English. **PRINT** accepts 3 print modifiers: Commas (,), Semicolons (;) and Apostrophes (').

Commas are used to make the printing start either at the left hand margin, or in the middle of the screen (depending on which comes next). Thus in line 50, the comma causes the Celsius temperature to be printed in the middle of the line. With a semicolon (;) on the other hand, the text number or string is printed immediately after the preceding one. You can see this in line 50: if the comma is replaced by a semicolon. Note here that this is the exact reason why we need to enter a colon before the semicolon if we need to use it as a **REM**Mark as discussed in the previous section!

Another punctuation mark you can use like this in **PRINT** commands is the apostrophe ('). This makes whatever is printed next appear at the beginning of the next line on the screen but this happens anyway at the end of each **PRINT** command, so you will not need the apostrophe very much. This is why the **PRINT** command in line 50 always starts its printing on a new line and it is also why the **PRINT** command in line 30 produces a blank line.

If you want to inhibit this, so that after one **PRINT** command the next one carries on on the same line, you can put a comma or semicolon at the end of the first! Let's see how this works: replace line 50 in turn by each of

```
50 PRINT F,
50 PRINT F
```

and

```
50 PRINT F'
```

and run each version — for good measure you could also try

```
PRINT F'
```

The one with the comma spreads everything out in two columns, that with the semicolon crams everything together, that without either allows a line for each number and so does that with the apostrophe — the apostrophe gives a new line of its own, but inhibits the automatic one.

Remember the difference between commas and semicolons in **PRINT** commands, also do not confuse them with the colons (:) that are used to separate commands in a single line. Now type in these extra lines:

```
100 REM this polite program
    REM remembers your name
110 INPUT N$
120 PRINT Hello ",N$,
130 GO TO 110
```

This is a separate program from the last one, but you can keep them both in the computer at the same time. To run the new one, type

```
RUN 100
```

Because this program inputs a string instead of a number, it prints out two string quotes. This is a reminder to you, and it usually saves you some typing as well. Try it once with any alias you care to make up for yourself!

Next time round, you will get two string quotes again, but you don't have to use them if you don't want to. Try this for example: Run Rem 20 (with ⇨ and DELETE keys) and type

```
N$
```


Since there are no string indices, the INPUT always reads a whole line as a single entity. In the calculation in this case, since the value of the string variable called n\$, which is whatever name you happen to have typed in last time round. Of course, the INPUT statement acts like **LET n\$=n\$** so the value of n\$ is unchanged.

The next time round, for comparison, type

```
n$
```

again. This time without hitting the return key. Now, since you haven't changed the variable n\$, the value "n\$" is printed.

Generally speaking, the INPUT parser is very intelligent with the calculation of the line above. There is no way to insert an invalid or improperly formed entry during INPUT. For example moving the cursor back to the beginning of the line using w and deleting the rest of the line with a cursor, ENTER will not do it. How artificial does it sound and the garbage will continue blinking until you correct the error by adding the closing quotes again.

Now look back at that **RUN 100** we had earlier on. That just jumps to the 100 statement. We could have said **GO TO 100** instead. In this case, so happens that the answer is yes, but the difference **RUN 100** instead of **GO TO 100** will lead all the variables around since the latter that works just like **GO TO 100**.

GO TO 100 does not do anything. There may well be occasions where you want to in a program without creating any variables. Here **GO TO** would be necessary and **RUN** could be disastrous, so it is better to get into the habit of automatically typing **RUN** to run a program.

An important reference is that you can type **RUN** without a line number, and it starts off at the first line in the program. **GO TO** must always have a line number or label.

Sometimes, by mistake, you write a program that you can't stop and won't stop itself. Type

```
10 GO TO 10
RUN
```

This looks all set to go on forever unless you pull the plug out, but there is a most drastic remedy. Press the **BREAK** key. The program will stop, saying **L BREAK** into program. At the end of every statement, the program looks to see if these keys are pressed and if they are then it stops. The **BREAK** key can also be used when you are in the middle of using the computer to enter a value. If you enter a value that is not intended, you can abort the computer, just in case the computer is waiting for you to do something but they are not doing. In those cases, there is a different word **DBREAK**. **CONT** repeats **CONTINUE**. In this case, after the program has finished, it repeats the statement where the program was stopped, but after the reports **L BREAK** into program or **9 STOP**. Statement **CONTINUE** always starts right on with the next statement after allowing for any jumps you made.

Run the same program again and when it asks you for input type

```
n$ (after removing the quotes.
```

n\$ is as of this moment an undefined variable and the computer will stop as it doesn't know what the value is. Computer will use **BREAK** to get out of the program and then type


```
n$='something definite'
```

(which has its own report of 0 OK, 01 and

```
CONTINUE
```

you will find that you can use `n$` as input data without any trouble

In this case **CONTINUE** does a jump to the **INPUT** command in line 10. It disregards the report from the **LET** command in this case, statement because that said OK and jumps to the command referred to in the previous report, the first format in line 10. This is intended to be useful. If a program stops over some error then you can do all sorts of things to fix it, and **CONTINUE** will still work afterwards.

As we said before, the report **2 BREAK into program's space** because after **CONTINUE** does not repeat the command where the program stopped.

We've seen so far programs where execution jumps to the beginning with no graceful way of ending the program. What we're producing are called *never ending loops* and are some of the great faults of programming. At all times there are some cases where execution can not be stopped. If for example we have disabled error reporting, or the **BREAK** key is inhibited, in these cases we have a program with either a dead loop or a program that uses a special keyword that ends a program prematurely and that keyword is **STOP**. Let's modify our polite program to be as follows:

```
10 REM this polite program
   remembers your name
20 INPUT n$
   PRINT 'Hello ',n$, ' '
   STOP
```

and then give **RUN**. After we enter our name and the computer greets us, we'll get a **9 STOP** statement in line 130, indicating we exited the program forcibly by the **STOP** command in line 10. We didn't leave line 130 entirely, and the program would have terminated with a 0 OK, 120 1 which would have indicated a proper program termination. In general, it's a good idea to provide exit paths in situations where the program may end up in a never ending loop. NextBASIC provides us with such facilities as we're going to see further on.

Error trapping

As we saw above, NextBASIC can occasionally generate error reports whether we have inadvertently caused them ourselves or because something is just wrong. Sometimes we need our program to stop execution and other times we want to recover from the error and continue as in the case above where we gave no **CONTINUE** command. For these cases, NextBASIC provides us with the **ON ERROR** command.

This can intercept (trap) any error report (except 0 OK which is not considered an error), thus allowing your programs to recover from expected error conditions.

Turning on error trapping is as simple as

ON ERROR statement(s)

This will cause the statement(s) immediately following the **ON ERROR** command to be executed whenever an error report would normally have been displayed. Note that this command must be part of a program and cannot be entered as a direct command.

To turn off error trapping again, just use **ON ERROR** on its own.

This is required if you wish to generate errors again (and you may wish to do so if you need to know what went wrong). The following example will display **There was an error** and terminate with the **9 STOP** statement error when line 20 is executed:

```
10 ON ERROR PRINT 'There was
   an error' ON ERROR STOP
20 PRINT 5/0
```

ERROR (n)

To generate the last error that actually occurred (this does not need error trapping to be turned off) just type the command:

ERROR

followed by **ENTER**. Assuming the program above, the following amendment will print the message but still give the correct **Number too big** report:

```
10 ON ERROR PRINT 'There was
   an error' : ERROR
20 PRINT 5/0
```

Used with the optional parameter *n*, where *n* is a value between 0 and 3, **ERROR** can return the error code (for *n*=0), the line (for *n*=1), the statement (for *n*=2) and the memory bank where it occurred (for *n*=3). See Chapter 23 for details about memory banks. For example:

PRINT ERROR ()

given after the example above would return 20. Moreover there's also

ERROR\$

which prints the error report rather than just the code. You could modify the example above to be

```
10 ON ERROR PRINT 'There was
   error' ", ERROR$ ON
   ERROR STOP
20 PRINT "
```

which will print the actual error report **Number too big**. You could then substitute **STOP** with a **GO TO** to the line of error handling code without having to halt execution of your program.

An additional way with which you can obtain details on the last error and store them away maybe for purposes of statistical analysis is using the following command:

ERROR TO codevar [, linevar [, statementvar [, bankvar [, ...]]]

This will store the error code in the numeric variable *codevar*, the line number in *linevar*, the statement number in *statementvar* and the bank number in *bankvar* (do not worry about what bank means for the moment). Note that you do not need to supply later variable names if you do not need the information, so all of these are valid:

```
ERROR TO e
ERROR TO e,l
ERROR TO e,l,s
ERROR TO e,l,s,b
```


For example, to get and store the error number in a variable `e` and then print it but still stop execution, we could modify the first program as follows:

```
10 ON ERROR PRINT 'There was
   an error ' : ERROR TO e
   PRINT e : ON ERROR STOP
   PRINT S/
```

If we allow the program to finish and then use **ERROR** we would have gotten the 9 **STOP** statement 10's error report which would be the last error report in statement 5 or line 10 as **STOP** is considered an error. But by using **ERROR TO** we'll get 6 printed on screen which is the error code for the **Number too big** error.

So far we have seen the keywords **PRINT**, **LET**, **INPUT**, **RUN**, **LIST**, **GO TO**, **CONTINUE**, **STOP**, **ON ERROR**, **ERROR**, **ERRORS**, **ERROR TO**, **NEW** and **REM**. Apart from **ON ERROR** you can also enter them as direct commands – this is true of almost all commands in NextBASIC. **RUN**, **LIST**, **CONTINUE** and **NEW** are not usually of much use in a program, but they can be used regardless.

Exercises

1. Put a **LIST** statement in a program so that when you run it, it lists itself.
2. Write a program to input prices and print out the tax due (a 20 per cent). Put in **PRINT** statements so that the computer announces what it is going to do, and asks for the input price with extravagant politeness. Modify the program so that you can also input the tax rate (to allow for zero ratings or future changes).
3. Write a program to print a running total of numbers you input. (Suggestion: have two variables called `total` (set to 0 to begin with) and `item`. Input `item`, add it to `total`, print them both, and go round again.)
4. What would **CONTINUE** and **NEW** do in a program? Can you think of any uses at all for this?

Chapter 2 Decisions

Making decisions

All the programs we have seen so far have been predictable. They went straight through the instructions then went back to the beginning again. This is not very useful in practice: the computer would be expected to make decisions and act accordingly. There are three ways *NexBASIC* helps you make decisions. The first is by using the **IF** keyword in a short, medium and long format. The second by using the **ON** case selection keyword and the third is by using the *select operator* ? (question mark).

Using **IF** to make decisions

The short and medium form of **IF** are as follows:

IF condition **THEN** action [**ELSE** alternative action]

Clear the previous program from memory by using **NEW** and type in and run the following example:

```
10 REM Guess the number
20 INPUT "Enter the number to
   guess", a :CLS
30 INPUT "Guess the number",
   b
40 IF b=a THEN PRINT "That is
   correct" :STOP
50 IF b<a THEN PRINT "That is
   too small, try again"
60 IF b>a THEN PRINT "That is
   too big, try again"
70 GO TO 30
```

You can see that in its simplest form an **IF** statement is

IF condition **THEN** action

where action stands for a sequence of commands separated by colons in the usual way. The condition is something that is going to be worked out as either true or false. If it comes out as true then the statement is if the rest of the line after **THEN** are executed, but otherwise they are skipped over and the program executes the next instruction.

The simplest conditions compare two numbers or two strings: they can test whether two numbers are equal, or whether one is bigger than the other, and they can test whether two strings are equal, or 'roughly' one comes before the other in alphabetical order. They use the relations = < > <= >= and <>. Additionally we can use **NOT**, **AND**, **OR**, ^ (bitwise NOT), | (bitwise OR), & (bitwise AND), ^ (bitwise XOR) all of which produce a true (1), or false (0) result.

= means equals. Although it is the same symbol as the = in a **LET** command, it is used in quite a different sense.

< means is less than so that

```
1 < 2
2 < 1
3 < 1
```

are all true, but


```
1 < 0
0 < 2
```

are false

> means *is greater than* and is just like **<** but the other way round. You can remember which is which, because the thin end points to the number that is supposed to be smaller.

<= means *is less than or equal to* so that it is like **<** except that it is true even if the two numbers are equal, thus $2 <= 2$ is true but $2 < 2$ is false.

>= means *is greater than or equal to* and is similarly like **>**.

<> means *is not equal to*, the opposite in meaning to **=**.

The remaining operators are explained in length in *Chapter 6 Expressions*.

Mathematicians usually write $<=$, $>=$ and $<>$ as \leq , \geq and \neq . They also write things like $2 < 3 < 4$ to mean $2 < 3$ and $3 < 4$ but this is not possible in *NexBASIC*.

Line 40 compares **a** and **b**. If they are equal, then the program is halted by the **STOP** command. The report at the bottom of the screen **9 STOP statement, 30:3** shows that the third statement (or command) in line 30 caused the program to halt – the **STOP**.

Line 50 determines whether **b** is less than **a**, and line 60 whether **b** is greater than **a**. If one of these conditions is true then the appropriate comment is printed, and the program works its way to line 70 which tells the computer to go back to line 30 and start all over again. The **CLS** command in line 20 clears the screen to stop the other person seeing what you put in.

Note: in some versions of BASIC the **IF** statement can have the form

```
IF condition THEN line number
```

This means the same as

```
IF condition THEN GO TO line number:label
```

ELSE

By adding the optional **ELSE** clause, more complex decisions can be made. This instructs the computer to run another set of commands if the **IF THEN** test turns out to be false. It is important to note that, unlike some other implementations of BASIC, **ELSE** must follow a colon within a statement line, for instance

```
IF number<0 THEN PRINT "Negative number"
ELSE PRINT "Positive number"
```

In the example above, if the condition is true (that is, the number is less than zero), then **Negative number** will be printed. Next, then **Positive number** will be printed on screen. But what if you, for example, wanted a third option to see if the number is zero? You could use the ability to "nest" **IF THEN** statements and use the **ELSE** clause to do so. Let's re-write the above:

```
IF number<0 THEN PRINT 'Negative number'
ELSE IF number>0 THEN PRINT 'Positive
number': ELSE PRINT 'The number is zero'
```

You should see in the above that it is possible to execute a further **IF THEN** statement if the condition in the original one was false. *NexBASIC* will work through the **IF THEN** statements until it finds a condition that is true, and will execute that. If no conditions are true, then it will attempt to execute the final **ELSE**. More than one command can be executed within each part of an **IF THEN ELSE** statement also, so


```

IF number < 0 THEN PRINT 'Negative number
GO TO 100 ELSE IF number > 0 THEN PRINT
'Positive number' GO TO 200 ELSE PRINT
'The number is zero' : zero += 1 GO TO
300

```

will allow you to jump to different parts of the program dependent on the results of the **IF THEN ELSE** statements in this case whether the number is negative, positive or zero (note that if the number is zero, one is added to the variable zero as well)

IF THEN ELSE however only works within a single program line and as a consequence it can be bulky and even somewhat unreadable. For that reason a longer form variation exists

IF condition [ELSE IF condition] action END IF

It is immediately apparent that there is no **THEN** clause. This is what determines if it's the long, medium or short form of the **IF** structure. Medium and short forms use **THEN** and are thus single program line structures whereas the absence of **THEN** makes it the long form one. The long form **IF** has some specific syntax requirements. First, off F condition needs to be the first statement on its line and so are **ELSE** and **ELSE IF**. Secondly, the whole **IF** structure's command sequence must be terminated by an **END IF** (which also needs to be the first statement on its line). Consider this example

```

100 IF x > 7 PRINT "x > 7
110 IF x > 1000
111 PRINT 'In fact it's
    huge'
112 ELSE
113 PRINT 'But not too
    big'
114 ENDIF
120 ELSE IF x > 3 PRINT 'x > 3 but
    x <= 7'
140 ELSE IF x = 3
150 PRINT 'x = 3'
160 ELSE
170 PRINT 'x is too small to
    bother'
180 ENDIF

```

It vividly demonstrates how a combination of nesting **IF ELSE ENDIF** structures within an external supersets **IF ELSE ENDIF** can allow multiple decisions in a relatively straightforward way. The main decision is contained in the initial **IF** on line 100 and the ensuing **ELSE** clause on line 160. There we're checking if *x* is greater than 7 or not. Obviously *x* can't be smaller than 7 but how small is immaterial? This gets asked by the **ELSE IF** clauses on lines 120 and 140 where it's decided what *x* or values from 3 (line 140) up to and including 7 (line 120) the value of greater than 7 has been dealt with on line 100 as we saw. It does indeed matter. There is however a set of values that *x* can have that is greater than 7 and that's what the nested **IF ELSE ENDIF** structure of lines 112 through 114 allows. It is to test again, where we further break it down to values greater than 1000 or less or equal to 1000 but still greater than 7. The **PRINT** statements are here to demonstrate, here's actions we can take per decision and could easily have been **GO TO** or **INPUT** or other sets of keywords actually doing something according to the value of *x*. Moreover, there are two styles of writing on display here. The compact one (lines 100 and 120) and the more relaxed with lots of whitespace one (lines 140 to 180).

Case selection with ON

Many BASIC dialects contain a special structure called **SELECT CASE** which is a specialised version of **IF...ELSE...ENDIF**. It's much easier to use and read. Next/BASIC has a somewhat similar keyword called **ON**. Its syntax is as follows:

```
ON n < stmt1 > < stmt1 > < stmt2 > ... < stmtm > ELSE < statements >
```

ON uses the resulting integer of rounding the value of expression **n** to the closest integer and executes the **nth** statement of the same line that follows it.

When the statement is executed, **ON** skips the rest of the line and moves to the next line. Unless the statement was a non-returning jump such as **GO TO**, **EXIT**, **RETURN** or **ENDPROC**. If **ON** runs out of statements (in other words the integer part of **n** is greater than the number of statements that follows it) then, if the optional **ELSE** clause is present, all the statements that follow **ELSE** are executed otherwise the entire line is skipped. For example:

```
100 ON X GO TO @xwaszero GO TO
    @xwasone STOP GOSUB 200
    ELSE BEEP 1,0 PRINT "x
    was > 3" STOP
110 PRINT 'execution
    returned' STOP
200 PRINT 'I'm in the
    subroutine now
210 RETURN
```

It's easy to understand that if **x** is 2 then the program will terminate. If it's 0 it will jump to label **@xwaszero**. If it's 1 it will jump to label **@xwasone**. If it was 3 it will jump to the subroutine on line 200, return and continue at line 110 and in every other case the computer will beep and inform us that **x** was greater than 3 before finally halting. While at first sight this may not display much improvement over the nested **IF...ELSE...ENDIF** structure especially since it can only check for values greater than 0, in reality however you can prepare the variable to be checked ahead of time and simply test for 0 (or 1) and moreover it is much simpler even typing-wise and more concise.

Decisions with the ? operator

A final way of making decisions on this time within the confines of a single statement that accepts a parameter is provided via the select operator **?** (question mark). This has the following form:

```
n? <exp1> <exp2> <exp3> ...
```

If this reminds you a bit of the **ON** keyword syntax above you'd be correct as based on the value of **n**, the **nth** expression on the list is evaluated. If you run out of expressions in the list then the last expression is always evaluated. The select operator can take either numeric (be it floating point or integer) or string values in its evaluation list however they all have to be the same. Although it's not immediately apparent how one could use it in the decision-making process type and **RUN** the following example which will make it all clear:

```
10 INPUT 'Enter a number ', n
20 PRINT n? (%5, %10, %20)
30 PRINT n? ('n is zero', 'n is
    one', 'n is two', 'n is
```



```
three", 'n is four or  
more')  
40 GO TO n7(100, 60, 120, 130,  
10)  
50 STOP  
60 PRINT 'n was one' : STOP  
100 PRINT 'n was zero' : STOP  
120 PRINT 'n was two' : STOP  
130 PRINT 'n was three' : STOP
```

(Do not mind the % symbols in the list of line 20: we will learn more about them in Chapter 6: *expressions*. As you enter values in the program, you will see how your input affects both the PRINT statements but also – and here's the important part – the GO TO statement on line 40. In effect creating a *conditional GO TO*. Moreover, instead of having to type 5 different GO TO statements like we would have in the case of IF...ELSE...END IF and ON...we only have to type one making it a great time-saver.

Obviously the ability of the *select* operator to take either *string* or *numeric* expressions as parameters makes it useful for many things like, for example a *conditional LET* apart from changing the flow of our program, so other than just line numbers we usually used, we can make it calculate and further enhance our decision making as we'll see in Chapter 6.

Chapter 3 Looping

Using FOR, TO and NEXT

Suppose you want to input five numbers and add them together. One way (don't type this in unless you are feeling dritful) is to write

```
10 total=0
20 INPUT a
30 total+=a
40 INPUT a
50 total+=a
60 INPUT a
70 total+=a
80 INPUT a
90 total+=a
100 INPUT a
110 total+=a
120 PRINT total
```

This method is not good programming practice. It may be just about controllable for five numbers, but you can imagine how tedious a program like this to add ten numbers would be, and to add a hundred would be just impossible.

Much better is to set up a variable to count up to 5 and then stop the program like this (which you should type in)

```
10 total,count=0,1
20 INPUT a
30 REM count=number of times
   that a has been input so
   far
40 total += a
50 count += 1
60 IF count <= 5 THEN GO TO 20
70 PRINT total
```

Notice how easy it would be to change line 60 so that this program adds ten numbers, or even a hundred.

This sort of counting is so useful that there are two special keywords to make it easier. **FOR** and **NEXT** that are always used together. Using these the program you have just typed in does exactly the same as

```
10 total = 0
20 FOR count = 1 TO 5
30   INPUT a
40   ,count=number of times
   that a has been input so
   far
50   total += a
60 NEXT count
```

80 PRINT total

The variable `total` is called the *control variable* of a **FOR** **NEXT** loop.

The effect of this program is that `count` runs through the values 1 (the initial value), 2, 3, 4 and 5 (the limit) and for each one, lines 30, 40 and 50 are executed. Then, when `count` has finished its five values, line 80 is executed.

STEP

The control variable does not have to increase by 1 each time; you can change this 1 to anything you like by adding a **STEP** clause in the **FOR** command. The most general form for a **FOR** command is

FOR control variable = initial value **TO** limit **STEP** step

where the initial value, limit and step are all *numeric expressions* (things in other words that the computer can calculate as numbers, like the actual numbers themselves, or sums, or the names of numeric variables). So, if you replace line 20 in the program by

```
20 FOR count=1 TO 5 STEP 3/2
```

then `count` will run through the values 1, 2.5 and 4. Notice that you don't have to restrict yourself to whole numbers, and also that the control value does not have to hit the limit exactly: it carries on looping as long as it is less than or equal to the limit. Try this program to print out the numbers from 1 to 10 in reverse order.

```
10 FOR n=10 TO 1 STEP -1
20 PRINT n
30 NEXT n
```

We have said before that the program carries on looping as long as the control variable is less than or equal to the limit. If you work out what this would mean in this case, you will see that it gives nonsense. The normal rule has to be modified: when the step is negative, the program carries on looping as long as the control variable is greater than or equal to the limit.


You must be careful if you are running two **FOR** **NEXT** loops together, one inside the other. Try this program, which prints out the numbers for a complete set of six spot dominoes.

```
10 FOR m=0 TO 5
20   FOR n=0 TO m
30     PRINT m," ",n
40   NEXT n
50 PRINT
60 NEXT m
```



You can see that the `n-loop` is entirely inside the `m-loop` — they are properly *nested*. What must be avoided is having two **FOR** **NEXT** loops that overlap without either being entirely inside the other, like this:

```
5 REM this program is wrong
10 FOR m=0 TO 5
20   FOR n=0 TO m
30     PRINT m," ",n;" ",
40   NEXT m
50 PRINT
60 NEXT n
```



Two **FOR** **NEXT** loops must either be one inside the other or be completely separate.

Another thing to avoid is jumping into the middle of a **FOR** **NEXT** loop from the outside. The control variable is only set up properly when its **FOR** statement is executed, and if you miss this out the **NEXT** statement will confuse the computer. You will probably get an error report saying **NEXT without FOR** or **Variable not found**.

Here is nothing whatever to stop you using **FOR** and **NEXT** in a direct command. For example try

```
FOR m=0 TO 10 PRINT m. NEXT m
```

You can sometimes use this as a (somewhat artificial) way of getting round the restriction that you cannot **GO TO** anywhere inside a command, because a command has no line number. For instance

```
FOR m=0 TO 1 STEP 0 INPUT a PRINT a
NEXT m
```

The step of zero here makes the command repeat itself forever.

This sort of thing is not really recommended, because an error crops up then you have lost the command and will have to type it in again, and **CONTINUE** will not work.

For additional speed and efficiency, *NextBASIC* also allows integer variables to be used as the index in **FOR** **NEXT** eg

```
10 FOR %I=%$CS TO 220
20 PRINT %I
30 NEXT %I
```

Integer loops run much faster than loops using a standard floating point control variable so they're preferred, especially where speed is a concern. Remember however that there is only a limited amount of integer variables, so a small bit of planning is warranted before starting with your program.

EXIT

Sometimes we need to prematurely exit from a loop, be it a **FOR** **NEXT** (See previous section) or a **REPEAT** **REPEAT UNTIL** (See next section). There is a seemingly obvious solution to that: jump out of the loop with **GO TO** however this is ill advised as **GO TO** doesn't exit a loop "cleanly" and therefore should not be used. Instead, here's a specialised command however that allows for a "clean" exit and that is (the aptly named)

EXIT [n]

where *n* is an optional line number or label to jump to. **EXIT** on its own will jump to the next statement after the end of the loop. Consider this example (the actual syntax of the loop is explained in the next section)

```
100 REPEAT
110 INPUT n
120 IF n=33 THEN EXIT 150
130 REPEAT UNTIL n<0
140 PRINT Loop ended normally
150 PRINT 'Loop ended early'
```

The loop above will terminate normally (when the condition set on line 130 is satisfied) when you enter a negative value but "early" if you input the value 33. **EXIT** can also be used to get out of nested loops using successive **EXIT** statements on the same line within

the innermost loop. Note that in such cases, only the final **EXIT** statement can take the optional parameter. To illustrate, consider the following example:

```
100 FOR i=1 TO 10
110   FOR j=1 TO 10
120     PRINT i,j
130     IF j*i>80 THEN EXIT EXIT 170
140   NEXT j
150 NEXT i
160 STOP
170 PRINT 'Product exceeded 80' STOP
```

Note that when in a loop that exists within a procedure and/or subroutine, it is acceptable to use **ENDPROC** or **RETURN** as a legitimate way to exit said loop.

REPEAT REPEAT UNTIL loops

NextBASIC has another way of looping: a set of commands for rather a single command block, called **REPEAT REPEAT UNTIL**. You will have noticed that **FOR NEXT** relies on counting to control the loop; however, you can also use a condition to control a loop. This type of loop begins with a **REPEAT** statement to indicate the beginning of the loop and a **REPEAT UNTIL** statement at the end, which also contains the condition to exit the loop. Try this:

```
10 REPEAT
20   INPUT 'Enter a number,
   or enter -1 to stop > ',n
30   PRINT n
40 REPEAT UNTIL n = -1
50 PRINT 'Thank you.'
```

This program will keep accepting numbers and printing them until you type -1 when it will politely thank you for your numbers. In a **REPEAT REPEAT UNTIL** loop, everything between the **REPEAT** and the **REPEAT UNTIL** command will be executed. In this case, this would be lines 20 and 30, until the condition in the **REPEAT UNTIL** statement becomes true (in this case, that the number you have entered is -1). Note that because the condition is checked at the end, the block of statements will always execute at least once. The following, for example, would print an erroneous statement:

```
10 x=1
20 REPEAT
30   PRINT "x is ",x;" but isn't 1"
40 REPEAT UNTIL x=1
50 PRINT "x is now 1."
```

Because line 30 is executed before the condition is checked at line 40, the message **x is 1 but isn't 1** will still be printed, which is clearly wrong. Like a **FOR NEXT** loop, you can also nest **REPEAT** loops if you need to. So:

```
10 n=1
20 REPEAT
30   PRINT "Counting to ",n
40   c=1
50   REPEAT
60     PRINT c," ", "
```



```

70      c+=1
80      REPEAT UNTIL c>n
90      PRINT "I'll count a bit
        higher"
100     n+=1
110     REPEAT UNTIL n=10
120     PRINT "OK, I'm done now"

```

will work fine. Try it and see if you can see what is happening. You can also make a **REPEAT** loop continue indefinitely if you use a zero in the **REPEAT UNTIL** statement. Type in this program:

```

10 REPEAT
20   PRINT "Hello world!"
30 REPEAT UNTIL 0

```

It will continue printing Hello world! on the screen, stopping only to ask if you want to scroll (unless you press the **BREAK** key, of course). Why? zero can be seen in NextBASIC as false when used in this way, so the **REPEAT UNTIL 0** statement will always give a false result, hence the loop will continue indefinitely. Obviously you can exit such a loop with **EXIT** if need be.

WHILE

The **WHILE** command, used within a **REPEAT** loop, can provide an alternative way of leaving the loop before reaching the **REPEAT UNTIL** statement. If the condition in the **WHILE** statement is true, the loop continues. But if it is false, then the remaining statements in the loop will be ignored, the loop will be exited and the program will resume with the line after the **REPEAT UNTIL** statement. Try this:

```

10 REPEAT
20   INPUT "Enter a number, or
        enter a negative number to
        stop > " n
30   WHILE n>=0
40     PRINT n
50 REPEAT UNTIL 0
60 PRINT "Thank you "

```

It is a different approach to the example seen earlier, this time using **WHILE** to check the number entered, and also accepting any negative number to stop. **WHILE** can also be used to exit a loop before any statements are executed, should you need to. Try

```

10 y=0
20 REPEAT . WHILE y<22
30   PRINT AT y,0,"This is
        line " y,"."
40   y+=1
50 REPEAT UNTIL 0

```

You will note that when *y* reaches 22, the loop will exit before printing the line number. It should also be pointed out that not only can you place a **WHILE** anywhere within the loop, but you can also place more than one **WHILE** in the same loop, if you have different conditions to check to leave the loop.

Error trapping within REPEAT / REPEAT UNTIL loops

Error trapping within REPEAT / REPEAT UNTIL loops as well as within subroutines and procedures is introduced. Refer to the next section of Chapter 4 on advanced error trapping for a complete example that covers all cases of error trapping in these programming structures.

Exercises

1. A **count** variable has not just a name and a value like an ordinary variable, but also a limit, a step, and a reference to the statement after the corresponding FOR statement. Persuade yourself that when the FOR statement is executed all this information is available using the initial value as the first value the variable takes and also that this information is enough for the NEXT statement to know by how much to increase the value, whether to jump back, and so, where to jump back to. Run the third program above and then type

```
PRINT count
```

Why is the answer **6** and not **5**? Answer: the NEXT command in line 60 is executed five times, each time adding to count, the last time count becomes **6** and then the NEXT command decides not to loop back, but to carry on, count being past its limit.

2. What happens if you put **STEP 2** in line 20?
3. Change the third program so that, instead of manually adding five numbers, it asks you to input how many numbers you want adding. When you run this program, what happens if you input 0 (meaning that you want no numbers adding)? Why might you expect this to cause problems for the computer, even though it is clear what you mean? (The computer has to make a search for the command **NEXT count** which is not usually necessary, in fact this has all been taken care of.)
4. In line 10 of the fourth program above, change ***0** to ***00** and run the program. It will print the numbers from 00 to 79 on the screen, and then say scroll? at the bottom. This is to give you a chance to see the numbers that are about to be scrolled off the top. If you press **BREAK** or the space bar, the program will stop with the report **0 BREAK**. **CONT** repeats, if you press any other key, then it will print all other 22 lines and ask you again.
5. Delete line 30 from the fourth program. When you run the new curtailed program, it will print the first number and stop with the message **0 OK**. If you type

```
NEXT n
```

The program will go once round the loop, printing out the next number.

6. Refer back to the example in the REPEAT UNTIL section where the message **x is 1** but it isn't 1 was displayed incorrectly. Rewrite this using **WHILE** so that the message does not appear when **x** is indeed 1. Change the value of **x** to 0, to check this works correctly.

Chapter 4 Procedures and Subroutines

Branching

As we already saw in the course of a program we may need to jump somewhere else inside the program. So far we've seen the **GO TO** keyword that does just that. **GO TO** however has disadvantages. If the code jumped to by **GO TO** needs to be used by some other portion of the program and then return to where it was called, you would need to either introduce complex code to find where the jump came from as to use an appropriate **GO TO** command to return to it or copy the code multiple times to be called individually by each location.

GO SUB and RETURN

To account for this deficiency, *NextBASIC* has the statements **GO SUB** *GO to Subroutine* and **RETURN** which are used together. Reusable code called upon multiple times within a program is known as a *subroutine* and it can be used, or *called*, from anywhere else in the program without having to type it again or remember where the call came from. The call to a subroutine takes the form

GO SUB n

where *n* is the line number or label of the first line in the subroutine. It is just like **GO TO n** except that the computer remembers where the **GO SUB** statement was so that it can come back again after doing the subroutine. It does this by putting the line number and the statement number within the line (together these constitute the *return address*) on top of a pile of them (the *NextBASIC return stack*—see Chapter 23 for details).

The command

RETURN

takes the top return address off the **GO SUB** stack and goes to the statement after it. As an example, let's look at the number guessing program again. Retype it as follows:

```
10 REM A rearranged guessing
   game
20 INPUT a:CLS
30 INPUT "Guess the number ",b
40 IF a=b THEN PRINT
   "Correct":STOP
50 IF a<b THEN GO SUB 100
60 IF a>b THEN GO SUB 100
70 GO TO 30
100 PRINT "Try again"
110 RETURN
```

The **GO TO** statement in line 70 is very important because otherwise the program will run on into the subroutine and cause an error (? **RETURN** without **GO SUB**) when the **RETURN** statement is reached.

Here is another rather silly program illustrating the use of **GO SUB**:

```
100 x=10
110 GO SUB 500
120 PRINT x
130 x+=4
```



```

140 GO SUB 500
150 PRINT s
160 x+=2
170 GO SUB 500
180 PRINT s
190 STOP
500 s=0 ; This is the start
    of the subroutine
510 FOR y=1 TO x
520   s+=y
530 NEXT y
540 RETURN

```

When this program is run, see if you can work out what is happening. The subroutine starts at line 500.

A subroutine can happily call another, or even itself: a subroutine that calls itself is recursive, so don't be afraid of having several layers.

LOCAL keyword

LOCAL is a special keyword reserved only for subroutines (see above) and procedures (see below) which ensures that the variables that follow it are independent of the rest of the program and only valid for the duration of the execution of the subroutine or procedure. Its syntax is as follows:

LOCAL var = value

where var is any variable type (string, numeric or integer) or array and value is the initial (or default) value for said variable. Local arrays need to be dimensioned separately after being localised by **LOCAL**. Multiple variables can be declared in the same **LOCAL** statement, separated by commas and there can be any number of **LOCAL** statements in a subroutine or procedure as long as there is enough memory for them. Consider the following example (which won't make much sense until you reach the *procedures* section further below):

```

100 PROC x(s(),size,2 TO
    result PRINT result STOP
110 DEFPROC x(input(),size,y)
120   LOCAL z() LOCAL tot=0
130   DIM z(size)
140   FOR i=1 TO size,z(i)=input(i)+
    i NEXT i
150   FOR i=1 TO size tot+=z(i) NEXT i
160   ENDPROC = tot

```

The moment that branching back occurs, local variables are released. Consider this silly example (which also demonstrates again the initialisation of local a\$):

```

10 a$ = "Test"
20 GO SUB 100
30 PRINT a$

```

Except integer arrays which are pre-mentioned


```

40 STOP
100 LOCAL a$="Different Value"
110 PRINT a$
120 RETURN

```

This will print Different Value and Test on your screen as LOCAL creates in a sense two versions of a\$: the second one exists only until the RETURN keyword is reached.

PRIVATE and PRIVATE CLEAR

PRIVATE is similar to LOCAL in that it initially reserved only for subroutines (see above) and procedures (see next section). Its syntax is as follows:

```
PRIVATE var1=value1
```

where var₁ is a numeric variable. Unlike LOCAL, only numeric variables are accepted by PRIVATE and value₁ is the optional initialisation value. The latter is important as you will see below. PRIVATE ensures that the variables that follow it are both independent of the rest of the program and that they are reinitialised every time the procedure is called. In order to reset the value of all private variables (to 0 or to the value initialised to by the PRIVATE statement) we can use PRIVATE CLEAR. When called from within banked code (See Chapter 23 - The Memory for information about banks and banked code) PRIVATE CLEAR initialises only the private variables of the procedures in that bank otherwise the private variables of the main program are reset. A small example of how PRIVATE and PRIVATE CLEAR work is the following:

```

100 PRIVATE CLEAR
110 FOR i=1 TO 10 PROC
    iterate, NEXT i STOP
120 DEFPROC iterate )
130 PRIVATE callcount=0
140 callcount+=1
150 PRINT "I have been called
    ",callcount,"times"
160 ENDPROC

```

Procedures (DEFPROC, ENDPROC, PROC)

Procedures are a special form of subroutines. Imagine them as a combination of subroutines and functions (See Chapter 8 - Functions). Like subroutines and the GO TO keyword they branch execution to a different segment of the program to better organise and reuse code. Unlike subroutines but like functions, they can accept multiple variables as parameters, can be named and when called they do not require a line number or label. Think of procedures as a way to extend NextBASIC commands much like user defined functions extending the built-in functions (see Chapter 8 for more details).

Procedure parameters can be regular numeric, integer and string variables as well as arrays which follow all the naming conventions of the former. As seen in Chapters 7 - Basic Programming Concepts, 7 - Expressions and 11 - Arrays.

Procedures can carry meaningful names following the naming conventions of numeric variables (See Chapter 7 - Expressions) or valid numeric variable names while they can also optionally be followed by a \$ sign. The latter has no effect in functionality but can be used to identify what the procedure does (for example manage strings).

Procedures are defined by the keywords DEFPROC which takes the form

```
DEFPROC name1$(, parameter1=def1, parameterN=defN)
```


and **ENDPROC** which takes one of two forms

ENDPROC

or (optionally)-

ENDPROC [=result1[... resultN]

Anything that follows the keyword **DEFPROC** is the procedure itself. However there can be multiple exit points for each procedure designated by separate **ENDPROC** statements.

Parameters in brackets denote that the syntax is optional and def1 to defN (also in brackets) are optional default values for each parameter. Procedures are called with the keyword **PROC** and **BANK PROC** in the case of a banked procedure. This (like **ENDPROC**) takes two forms

[BANK n] PROC name ([parameter1[... [parameterN]])

which calls the procedure named name with optional parameters through N or-

'BANK n' PROC name ([parameter1[... [parameterN] ?] **TO** variable1[, variableN] which is the same as above but assigns the values returned by the procedure to the optional variables 1 through N.

Note that if there's a default value for a parameter we can omit it when calling the procedure with **PROC**.

NOTES

If you're using the MexiBASIC's memory bank management facilities to extend the size of your programs, the following apply:

- 1 Any **GO TO PROC** or **GO SUB** within a banked section will go to where it is defined (the same bank).
- 2 Any **RETURN** will always return to the calling bank.

Consider the example below

```

10 CLS
20 PROC Pdemo(11) = PROC
   HelloWorld("Hello
   World", 1)
30 PROC HelloWorld("Hello
   Stop!", 2) , 2nd parameter
   omitted
40 GO TO 150
50 DEFPROC Pdemo(x)
60   PRINT x " raised to the
   2nd power is ", x*x
70 ENDPROC
80 DEFPROC HelloWorld(z$,
   n=0)
90   LOCAL a$,l

```



```

100 IF n=0 THEN PRINT z$
    ENDPROC
120 IF n=1 THEN LET l=LEN z$
130 a$=z$ \ +z$+z$ \
140 PRINT z$' INVERSE 1, a$
150 ENDPROC
160 STOP

```

As you can see, there's a default value for parameter *n* and two separate exit points for procedure **HelloWorld**, one at line 100 and one at line 150. Line 40 is mandatory or rather a condition to jump over the procedures defined is mandatory as without it, after execution of both procedures, the next available line would have been 50. **DEFPROC** can only appear in a program file. Attempting to define a procedure interactively will result in the error **Direct Command Error**. Also, supplying the wrong type of variable as a parameter (ie. a string instead of a number) will result in the error **Q Parameter error**.

Executing the program will return the following:

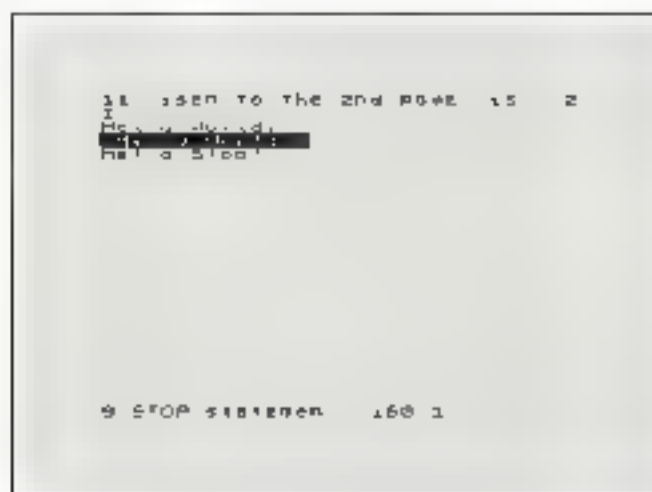


Fig. 3 Screen output from the sample procedures

As we saw in the definition of the **DEFPROC**, **ENDPROC** and **PROC** keywords as well as the examples above, there are optional parameters that can be passed to procedures when called with the results of the procedures' execution being assigned to multiple variables at the time. Consider this example that calculates the factorial of a number:

```

10 INPUT 'Enter a number'
   l+ ,x
20 IF x>23 THEN PRINT 'Your
   Next cannot handle this
   number ' GO TO 999
30 PROC factorial(x) TO f
40 IF f,0 THEN PRINT 'The
   factorial of ",x," is ",f
   ELSE GO TO 999
999 STOP
1000 DEFPROC factorial(n

```



```

1010 IF n<0 OR n<> INT n THEN
      PRINT 'Factorial only
      possible for 0 or positive
      integers' ENDPROC =
      1 ELSE IF (n = 0 OR n=1)
      THEN ENDPROC =1
1020 LOCAL partial
1030 PROC factorial(n 1, TO
      partial
1040 ENDPROC =nepartial

```

Apart from being a good example of recursion² we can see how this procedure feeds itself the results of the previous iteration via the local variable **partial**. Each iteration reduces the value by 1 as evidenced in line 1030. There's an obvious extra iteration that could be skipped when *n* becomes 1 but it's not important for the purpose of this example.

When calling a procedure with the PROC TO version of the PROC keyword ENDPROC must use the optional form ENDPROC =result1 and have as many results returned separated by commas as the calling PROC requested. PROC may be called without a TO or with a partial list of the result variables returned by ENDPROC but the inverse cannot happen and will return error Q Parameter error. For example this program

```

10 product = 0
20 PROC mul(3) TO product
30 PRINT product
40 STOP
50 DEFPROC mul(x)
60   LOCAL a
60   a=x*2
60 ENDPROC =a

```

will return 6 when run. When we change line 20 to read

```
20 PROC mul(3)
```

it will return 0 as variable **product** hasn't been changed from its initial assignment. However if we return line 20 to its original form and change line 70 to

```
70 ENDPROC
```

then execution of the program will produce a Q Parameter error.

Passing parameters by reference with REF

Normally calling a procedure with parameters is done *by value*. This means that the value of the parameter is passed on to the procedure and any changes that occur are invisible to the caller. Passing *by reference* on the other hand lets the caller be aware of the changes.

This is mainly intended for arrays as the default (by value) will be slower and more memory intensive but it can also be useful for strings. Numerical variables are faster if passed by value so passing them by reference is not recommended. To make a parameter a reference we use the REF keyword within the DEFPROC parameter block. Such parameters must be passed by the calling PROC as a variable or array name only. It is not permitted to have default values for REFerenced parameters. For example

²The ability of the code to call itself


```

100 PROC t(x$,5) STOP
110 DEFPROC t$(REF
    input$,index=1,
120 input$(index) =
    input$ index) (2 TO
130 ENDPROC

```

On the example above `index` is not passed by reference and therefore `t` can have a default value. Note that integer variables and arrays cannot be passed by reference.

Reading parameters with DATA and READ

A PROC may call a DEFPROC with more parameters than the DEFPROC requires. In this case if the DEFPROC has the keyword **DATA** as its final parameter, the procedure may read the additional parameters one at a time using **READ**. The new function **DATA** (see also Chapter 6) can be used to determine if there are further PROC parameters left to read. As an example:

```

100 PROC printer("Digits",0,1,2
    3,4 5,6,7,8 9) STOP
110 DEFPROC printer(name$, DATA
115     LOCAL n
120 PRINT name$
130     REPEAT WHILE DATA
140 READ n PRINT n
150 REPEAT UNTIL 0
160 RETURN

```

Trapping errors locally

As well as (or instead of) having a global error-trapping routine for your program as exhibited at the end of Chapter 1, each procedure, subroutine and repeat loop may have its own local error-trapping routine, simply by using the **ON ERROR** command within it.

When an error occurs within a repeat loop, subroutine or procedure, it will be trapped by its own **ON ERROR** routine if there is one. If not, the error will be passed out to the next level and trapped by any **ON ERROR** routine there and so on. Only if there is no **ON ERROR** at any level above the command that caused the error will a formal error report be generated. For example:

```

10 ON ERROR PRINT "Outer error
    handler:" ERROR
20 REPEAT
30     PRINT "Starting..."
40     ON ERROR PRINT "Oops!" ON
        ERROR STOP
50     GO SUB 100
60     PRINT "Iterating..."
70     ON ERROR
80 REPEAT UNTIL 0
90 STOP
100 ON ERROR PRINT "Bad
    pigs" RETURN

```



```
110 PROC myproc()  
120 PRINT 'Pigs ',pigs  
130 RETURN  
200 DEFPROC myproc()  
210   LOCAL m  
220   ON ERROR PRINT "Myproc  
    died..." ENDPROC  
230   PRINT "m=",m, "n=",n  
240 ENDPROC
```

Note that any `LOCAL` commands in a procedure or subroutine must come before a local error handler (ie lines 210 and 220 in the example cannot be reversed)

Chapter 5 READ, DATA, RESTORE

READ, DATA and RESTORE

In some previous programs we saw that information (or data) can be entered directly into the computer using the **INPUT** statement. Sometimes this can be very tedious, especially if a lot of the data is repeated every time the program is run. You can save a lot of time by using the **READ**, **DATA** and **RESTORE** commands. For example

```
10 READ a,b,c
20 PRINT a,b,c
30 DATA 10,20,30
```

A **READ** statement consists of **READ** followed by a list of the names of variables, separated by commas. It works rather like an **INPUT** statement, except that instead of getting you to type in the values to give to the variables, the computer looks up the values in the **DATA** statement.

Each **DATA** statement is a list of expressions (numeric or string expressions, separated by commas). You can put them anywhere you like in a program, because the computer ignores them except when it is doing a **READ**. You must imagine the expressions from all the **DATA** statements in the program as being put together to form one long list of expressions, the **DATA** list. The first time the computer goes to **READ** a value, it takes the first expression from the **DATA** list, the next time it takes the second, and thus as it meets successive **READ** statements, it works its way through the **DATA** list. (If it has to go past the end of the **DATA** list, then it gives an error. See the next section for an easy way to avoid that.)

Note that it's a waste of time putting **DATA** statements in a direct command, because **READ** will not find them. **DATA** statements have to go in the program. Let's see how these fit together in the program you've just typed in. Line 10 tells the computer to read three pieces of data and give them the variables **a**, **b** and **c**. Line 20 then says **PRINT** these variables. The **DATA** statement in line 30 gives the values of **a**, **b** and **c**. To see the order in which things work change line 20 to

```
20 PRINT b,c,a
```

The information in **DATA** can be part of a **FOR...NEXT** loop. Type in

```
10 FOR n=1 TO 5
20 READ d
30 DATA 2,4,6,8,10,12
40 PRINT d
50 NEXT n
```

When this program is **RUN** you can see the **READ** statement moving through the **DATA** list. **DATA** statements can also contain string variables. For example

```
10 READ date$
20 PRINT 'The date is ',date$
30 DATA 'January 1st, 2024'
40 STOP
```

It is as the simple way of fetching expressions from the **DATA** list, start at the beginning and work through until you reach the end. However, you can make the computer jump about in the **DATA** list using the **RESTORE** statement. This has **RESTORE** followed by a line number, and makes subsequent **READ** statements start getting their data from there.

first **DATA** statement) at or after the given line number. (You can miss out the line number in which case it is as though you had typed the line number of the first line in the program.)

Try this program:

```
10 READ a,b
20 PRINT a,b
30 RESTORE 10
40 READ x,y,z
50 PRINT x,y,z
60 DATA 1,2,3
70 STOP
```

In this program the data required by line 10 made $a=1$ and $b=2$. The **RESTORE 10** instruction allowed x , y and z to be **READ** starting from the first number in the **DATA** statement. RUN this program again, without line 30 and see what happens.

NOTES

You can store **DATA** statements in memory banks – take advantage of the expanded memory available on the ZX Spectrum. Now refer to Chapter 23 – The Memory – for information on how to use **BANK RESTORE** to do this.

READ, **DATA** and **RESTORE** accept integer variables following the conventions set forth in Chapter 5 – Expressions.

DATA (function)

Apart from its use in subroutines as we saw in the previous chapter, **DATA** can also be used as a function that returns what the next **DATA** item to be read is. Return values can be numeric (1), string (2) or none (0) when there are no more data items available. This is most useful if we want to load a list of items with varying data types. Consider this example:

```
10 END ON DATA GO TO @prn
   READ number READ text@
   ELSE GO TO @rd
20 GO TO @rd
30 @prn PRINT text$ number
40 DATA 3,"Here we come KS"
```

What have we done here? We've used the handy **ON ELSE** conditional selection structure and placed the **DATA** function in lieu of a variable argument since it returns one of 3 values. Then used the data type look ahead that **DATA** as a function provides to decide whether our next **READ** operation will be a numeric or string one. As **ON** expects its statements according to an increasing order of values, the exit condition is first.

Note that **ELSE** is really redundant here as there are 3 possible values of **DATA** and all are covered by the statements available. However it was put there as an additional demonstration of the complete **ON ELSE** structure.

Chapter 6 Expressions

Mathematical operations +, -, *, /, MOD

You have already seen some of the ways in which the ZX Spectrum Next can calculate with numbers. It can perform the four arithmetic operations +, -, *, / and remember that * is used for multiplication and / is used for division and can find the value of a variable given its name. The example

```
TAX=SUM*20/100
```

gives just a hint of the very important fact that these calculations can be combined. Such a combination like `SUM*20/100` is called an expression, so an expression is just a short-hand way of telling the computer to perform several calculations in parallel. In our example the expression `SUM*20/100` means *look up the value of the variable called SUM, multiply it by 20, and divide the result by 100*.

There's also one more mathematical operation, the modulo which returns the remainder of a division. It is used in the same way as the division operator but is denoted instead by MOD. As an example the direct commands

```
PRINT %17 MOD 6  
PRINT 17 MOD 6
```

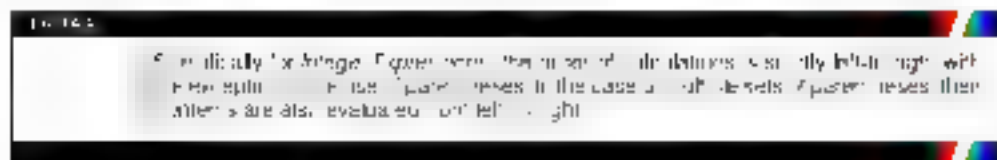
will both return 5 which is the remainder of the division of 17 by 6. Note the percent symbol % that precedes 17 in the first example, this is what defines it as an integer Expression. We will look at this in a little bit.

Order of mathematical calculations

It's important here to underline the order in which mathematical expressions are evaluated. Multiplication, division, and modulo are more important than addition and subtraction. For example, multiplication and division have the same priority, which means that in `2*3+4` the multiplications are done first, then the addition. So right when they are dealt with the additions and subtractions come next. These again have the same priority as each other, so we do them in order from left to right. This is very important because sometimes it is a bit tricky to write exactly the order.

Although all you really need to know is whether one operation has a higher priority than another, the computer does this by having a number between 1 and 16 to represent the priority of each operation. * and / have priority 8 and + and - have priority 6.

This order of calculations is absolutely rigid, but you can circumvent it by using parentheses. Anything in parentheses is evaluated first and then treated as a single number.



Bitwise, relational and logical operators

With every kind of expression there's a number of bitwise, relational and logical operations that can be performed other than simple mathematical operations. Specifically, on numbers these can be performed on floating point integers. While the operators are the same, there are subtle and some not so subtle differences in the way things work and we'll attempt to show these below.

In addition to the above there's a unary operator which follows

Unary/Bitwise NOT (!)

As we discussed in the previous section NextBASIC provides one additional unary operator which is an operator that requires one number alone. This is

! bitwise NOT

Bitwise NOT inverts the bits of said number from 0 to 1 and vice-versa.

<code>PRINT %!15</code>	returns 65520 as 15 gets inverted to become 65520	(0000 0000 0000 1111 1111 1111 1111 0000)
<code>PRINT %!43690</code>	returns 21845 as 43690 gets inverted to become 21845	(1010 1010 1010 1010 0101 0101 0101 0101)

this seems pretty straightforward correct? Wrong because look at what happens once you omit the % symbol and the expression is no longer an integer one

<code>PRINT !43690</code>	returns -43691 as 43690 gets inverted together with its sign bit to become -43691	(1010 1010 1010 1010 0101 0101 0101 0101)
---------------------------	---	--

what happened can be found in a simple phrase: two's complement which we'll examine further below. Until then however run the following examples

```
10 PRINT 32767
20 PRINT %! 32767
30 PRINT % SGN(!32767)
```

if you feel a bit overwhelmed not to worry soon all will be very clear

Bitwise operators <<, >> &, ^, ^

NextBASIC can also perform 5 bitwise operations (that is operations on the individual binary digits that make up a number or variables and expressions). These are:

<code>x << y</code>	Shift each bit of x y places left
<code>x >> y</code>	Shift each bit of x y places right
<code>x & y</code>	Bitwise AND between x and y
<code>x y</code>	Bitwise OR between x and y
<code>x ^ y</code>	Bitwise XOR between x and y (Deprecated, use the next one,
<code>x ^ y</code>	Bitwise XOR between x and y (Synonym with the previous)

More information on Bitwise operations together with examples (as binary examples are much easier to understand) can be found in *Integer Expressions* below. The operators however work on both (except for the deprecated one) Integer and Floating Point expressions

Logical operators

Standard logical operators can be used within integer expressions if prefixed by a % These are used in the same manner as their floating point counterparts

<code>x AND y</code>	Logical AND (gives 0 if y is zero, x if y is non-zero)
<code>x OR y</code>	Logical OR (gives x if y is zero, 1 if y is non-zero)
<code>NOT n</code>	Logical NOT (zero > 1 non-zero > 0)

Relational operators <, >, =, <=, >=, <>

<	less than
>	greater than
=	equal to
<=	less than or equal to
>=	greater than or equal to
<>	not equal to

Whenever you use them in the integer programming context, the six relational operators will always return the variable-like result of 0 for false and 1 for true.

Expressions

Expressions are useful because whenever the computer is expecting a number from you, you can write an expression instead and will work much more easily. The exceptions to this rule are so few that they will be stated explicitly in every case.

You can add together as many numbers and variables as you like in a single expression, so long as you write them all. Even use parentheses if you like. In integer expressions there are some other considerations and limitations as well as additional arithmetic and bitwise operators and functions, so they will all be separated from each other below.

Variable names and limitations

Basically, you can call your variables anything you like, but it isn't as free as the names of variables. As we have already said, the name of a string variable has to be allowed by \$, otherwise they are forbidden in the same way names of arrays and subscripts are if they contain the characters \$ or % or any other special characters as well as the characters \$ and %. They won't count as part of the name. Also, it doesn't make any difference if the name will be all upper or all lower case letters. The only sure rule is that all variable names will have the same as or more of the keywords we've introduced if the variable is a BASIC keyword. In all with spaces, either side, then it will not be accepted.

Integer variables are a bit different, as they can only be a single letter (A to Z or lower case a to z) and they're assigned in an expression that begins with a % eg

```
%a = 10
```

Actually, all integer variables are treated by default as unsigned, but values except when you use the special `SIGN()` keyword in which case they're signed. See the relevant section at the end of this chapter for details.

All calculations are performed with the numbers in floats, meaning all results are rounded to a maximum value of 65535 with 16 bits in the low 16 bits and 16 bits above by zero, which results in error 6. Number too big. Integer variables are pre-allocated and stored in a bit of memory outside the normal memory used by BASIC. This gives us a limit of 32768 integer variables as well as memory savings and the ability to use all binary numeric variables.

Further, you can embed a true arithmetic or logical expression in a variable and arrays contained within the same expression are integer ones. In cases where there is more than one integer expression within a line, each needs to be preceded with a %.

Here are some examples of the names of variables that are allowed

```
x
142
ItIsWithAHeavyHeartThatIMustSay
nowWeAreSix
```



```
nQWWeaReSjX
```

(these lgs. two names are considered the same and refer to the same variable)

The following are *not* allowed to be the names of variables

```
pi
```

PI is a keyword

```
2001
```

(it begins with a digit)

```
A new variable
```

(contains the separated keyword NEW)

```
3 bears
```

(begins with a digit)

```
M*A*S*H
```

(* is not a letter nor a digit)

```
Fotherington-Thomas
```

_ is not a letter nor a digit

Integer variables can only use the letters **A** to **Z** (again, case does not matter, so **a** or **z** are also acceptable), as you can see below, for a variable to be treated as integer, a % symbol somewhere in the same expression must precede it

Scientific notation

Numerical expressions can be represented by a number and exponent. Try the following to prove the point:

```
PRINT 2.34e0
```

```
PRINT 2.34e1
```

```
PRINT 2.34e2
```

and so on up to

```
PRINT 2.34e15
```

You will see that after a while the computer also starts using scientific notation. Similarly try

```
PRINT 2.34e-1
```

```
PRINT 2.34e-2
```

and so on

PRINT gives only eight significant digits of a number. Try

```
PRINT 4294967295,4294967295 429e7
```

This proves that the computer can hold the digits of **4294967295**, even though it is not prepared to display them all at once.

The ZX Spectrum Next, unless integer variables are expressly used (see above), uses floating point arithmetic, which means that it keeps separate the digits of a number, its mantissa, and the position of the point (the exponent). This is not always exact, even for whole numbers.

Type

```
PRINT 1e10+1-1e10,1e10-1e10+1
```

Numbers are held to about nine and a half digits accuracy, so **1e10** is too big to be held exactly right. The inaccuracy (actually about 21) is more than 1, so the numbers **1e10** and **e10+1** appear to the computer to be equal. For an even more peculiar example, type

```
PRINT 5e9+1 5e9
```

Here the inaccuracy in **5e9** is only about 1, and the 1 to be added on in fact gets rounded up to 2. The numbers **5e9+1** and **5e9+2** appear to the computer to be equal.

The largest integer which number that can be held completely accurately is 1 less than 32 2s multiplied together (or 4,294,967,295) — in other words $2^{32}-1$.

The string "" with no characters at all is called the empty or null string. Remember that spaces are significant and an empty string is not the same as one containing nothing but spaces.

```
PRINT 'Have you finished 'Finnegans Wake' yet?'
```

When you press ENTER you will get the flashing red cursor mark that shows there is a mis-ack somewhere in the line. When the computer finds the double quotes at the beginning of "Finnegans Wake" it imagines that these mark the end of the string "Have you finished " and it then can't work out what Finnegans Wake means.

There is a special device to get over this whenever you want to write a string quote symbol in the middle of a string you must write it twice like this:

```
PRINT 'Have you finished ""Finnegans Wake"" yet?'
```

As you can see from what is printed on the screen, each double quote is only really there once — you just have to type it twice to get it recognised.

Decimal, Binary and Hexadecimal numbers

Number literals in NextBASIC can be expressed as: *Decimal* (default), *Binary* (preceded by @) and *Hexadecimal* (preceded by \$). In the case of integer-only literals, the same rule as any with other integer expressions applies: if binary and hexadecimal literals they need to be preceded by %, once per expression. Consider these examples:

```
PRINT %E3, @11100011
PRINT $E3, %011100011
PRINT %E3+@11100011
PRINT $E3+%011100011
```

The first example prints an integer and then a floating point number which is converted from hexadecimal and binary respectively. The same thing happens in the second case but with the first value being a floating point one and no second an integer as again there are two separate expressions following the PRINT keyword. The third example is also valid as it contains a properly marked (preceded by %) integer expression. The addition of the hexadecimal and binary numbers is a single integer expression as the % preceding the addition marked both numbers as integer and therefore doesn't need a second %. The fourth example however is NOT valid as it's trying to add a floating point number with an integer number and this expression will fail.

In the case of floating point literals, both hexadecimal and binary numbers can have fractional parts. For example:

BIN 1.1	is the same as 1.5 (decimal)
@10.01	is the same as 2.25 (decimal)
\$64.C	is the same as 100.75 (decimal)

More about Integer Expressions and Variables

As previously mentioned, the main two reasons for the use of integer variables, Arrays and Expressions is memory efficiency and speed of execution.

Integer variables can be used in assignments using keywords **INPUT**, **LET**, **READ**, **FOR**, **ENDPROC** and **PROC** by preceding their name with a % symbol.

Normally it is not possible to access standard numeric variables or functions within an integer expression, or to access integer variables or operations within a standard numeric expression. In the following program

```
10 a=3
20 b=4
30 %a=2
40 %b=5
50 c=%a*b
60 d=%b*a
70 PRINT c,d
80 %b=b
90 c=%a*b
100 PRINT c,d
```

you might expect line 70 to produce 8 and 15. Instead it returns 10 and 10 as the % in lines 30 and 40 indicates that the entire expression is an integer expression, and all the variables named in each line are integer variables even though each name is not directly preceded by a % and only line 100 produces a different output: 8 and 10 respectively.

It is as apparent from the above example possible therefore to assign an integer expression to a standard normal numeric variable or vice-versa and the value will be converted appropriately. This automatic conversion is called *casting* and it's best illustrated in line 80 above as well as the examples below which are all valid assignments.

```
%A=2*PI*radius
```

assigns a truncated floating point calculation to integer variable A

```
%B=%B+(A(7)<<3)
```

shifts integer array element A(7) left 3 bits and adds it to integer variable B

```
addr=%X(1)<<8+X(0)
```

calculates standard numeric variable addr from low and high bytes in integer array X elements 0 and 1

As we saw earlier it's not normally possible to use a floating point expression within an integer expression. But what if we needed to do so? Consider the following example:

```
%a,%b %c=1 PRINT %a+PI+b+c
```

Looks simple enough, doesn't it? All we expect to happen is for casting to take over and use just the integer portion of the value of PI but it doesn't work that way. Instead the cursor flashes next to PI and the NextBASIC editor complains. To address this NextBASIC includes the special **INT {p expression}** keyword (do not omit the braces) which converts/casts any floating point expression {p expression} into an integer. So even if the example above wouldn't work, a small change

```
%a,%b %c=1 PRINT %a+INT { PI }+b+c
```

and it works happily. As a matter of fact, **INT { }** will convert any expression that produces a floating point value. Here are some examples:

```
test = 3.45 PRINT % INT {test}
```



```
alpha = 0 beta = 1 %a = %0111 + INT
{alpha OR beta}

%x=%x+INT{(INKEY$="P" OR
INKEY$="p")} INT{ INKEY$= 0 OR
INKEY$="o")}
```

Despite the presence of `INT{...}` in order to avoid confusion and unexpected results that can make debugging very hard, it would be a good practice to not use one or more single letter standard variables when there's a possibility of a similarly named variable existing in its integer form and instead use a more easily identifiable name.

We discussed about using floating point literals and/or expressions within the integer expressions evaluator, what happens when we want to do the opposite, to use in other words an integer expression as a sub-expression within the standard expression evaluator?

As it happens, this is possible as long as it is started after any opening parenthesis or separating comma. For example:

```
x%=STR$(%a, %b, 5)
x=apples+pears+(%x(3))
```

There is a notable exception to that requirement. For the new **BANK** functions (See *Chapter 23* for details about **BANK** in the standard expression evaluator) the bank number can be considered to be implicitly within parentheses, so it may be specified using an integer expression directly as follows:

```
x=2*PI%BANK %b PEEK myaddress
```

As we saw from the binary operator, bitwise operations on variables and arrays are pretty straightforward and involve manipulations of the individual bits of any number as represented in the ZX Spectrum Next's memory.

Shifting left or right involves moving the binary content of a variable `x` places (bits) to the left or right, padding from the right or left respectively with as many 0s as the places we shift the number for.

To illustrate bit shifting we can do the following example. Let's assign the decimal number 1201 first to an integer variable `A`, then manipulate its bits by shifting them left and right and printing the result so we can compare:

```
100 %A=1201
110 %A>>=3
120 %A<<=3
130 PRINT %A
```

This will return 1200 when run. To demonstrate what went on we could illustrate the expressions in two consecutive **PRINT** statements:

```
100 PRINT %1201>>3
110 PRINT %150<<3
```

Debugging is the programming process where you first attempt to reproduce if a program has errors, then identify these errors and finally to remove them!

Once we see how the numbers are stored in memory as a series of bits we can easily understand what happened:

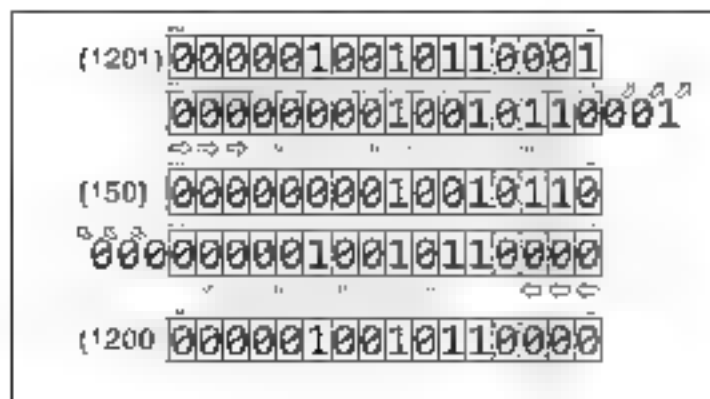


Fig. 4 Bit shifting

What happens however if we do the same to a floating point variable? Let's rewrite the above example

```
100 A=1201
110 A>>=3
120 A<<=3
130 PRINT A
```

which prints the same number as what we have assigned in line 100! Why the difference? We will need to recruit a function from a little further down this manual to help us better understand. Just type the following

```
100 A=1201 PRINT
    A,STR$(A,2,4)
110 A>>=3 PRINT A,STR$(A,2,4)
120 A<<=3 PRINT A,STR$(A,2,4)
```

The whole trick is in the fractional part of the number we don't normally see because it's 0. By shifting 3 places to the right we occupied 3 of the fractional places and therefore when we shifted back to the left the number wasn't truncated from the right side of its binary representation.

The remaining bitwise operations are very straightforward. Bitwise AND & is used to quickly determine if a bit inside a number is set to 1 or not. The first operand is the number we want to check and the second one is called the *bitmask* which is the number we check against. Consider these two examples

```
PRINT %010101010 & %01010101
PRINT %11100011 & %10
```

First example will return 0 while the second 2. The reason for this is that the numbers in the first example don't have coinciding 1 bits in the same positions while on the second example the second bit will be 1 and as a consequence the bits that match will be the first and second which make binary 10 which in decimal equals 2. To illustrate further

70		10101010
AND 85	(Bitmask)	01010101
Result		00000000

As you can see no bit set to 1 in any position of the two numbers matches each other therefore the result returned is 0 whereas in the second example

227		11100011
AND 2	(Bitmask)	00000010
Result		00000010

Bit 2 of the mask matches bit 2 of the number and it's 1 therefore 10 is returned (binary equivalent of decimal 2)

Bitwise OR will return 1 in any position if at least one bit of the two numbers in the same position is 1 and 0 if both are set to 0. For example

```
PRINT %010101010 + %01101011
```

will return 235 as only bits in positions 3 and 5 in both numbers are set to 0 making the resulting number 11101011 in binary form (or 235 in decimal). To better illustrate

70		10101010
OR 127	(Bitmask)	01101011
Result		11101011

Finally, bitwise XOR (^) will only return 1 in any position if either bit is set to 1 but not both. So two 0s and two 1s both return 0 in a position. Using the same numbers as in the previous example

```
PRINT %010101010 ^ %01101011
```

will return 193 or binary 11000001 since

70		10101010
XOR 127	(Bitmask)	01101011
Result		11000001

Bitwise expressions are uniquely helpful in determining the condition of flags in several of the ZX Spectrum Next ports (as we will see in *Chapter 22*) since these are the form of individual bits in a binary number and testing those with regular arithmetic can be cumbersome and slow.

Signed vs Unsigned Integer Expressions

As you saw in *Chapter 2* and in the introduction to this chapter, integer variables in NextBASIC are fixed to 16-bits wide *unsigned*, which means that they can display only positive integers from 0 to 65535. To illustrate approximately what that means, try the following

```
PRINT %-64448
```

The computer will respond with 1088. Keep this result in mind for a moment and then try

```
PRINT % 32448
```

This time the computer will display the number 33088 on screen. Are you confused yet? Maybe seeing the numbers in binary will help. Let's start with the first response of 1088 and we'll work backwards.

Decimal	Binary
1088	0000 0100 0100 0000
-64448	1 0000 0100 0100 0000
64447	1111 1011 1011 1111

Don't mind this for now!

Aha! Let's now see the second response

The computer returns:

```
64447      32447
```

And if we add these together by doing

```
PRINT %1088 + 64447, %33088 + 32447
```

we will get in both cases 65535!

Integer arithmetic is extremely fast, so we should have at least a way of representing signed integers in NextBASIC for both fast calculations as well as special cases, so NextBASIC does provide the way to deal with those numbers with the special **SGN { }** keyword. What this does is to treat any integer expression enclosed within it as a signed integer value (ranging from -32768 to 32767). All expressions enclosed within an **SGN { }** block are called *signed integer expressions*. Signed integer expressions use all the same operators and functions as standard unsigned ones, but the arithmetic operators (+ * MOD) and the relational operators (< <= > >= = <>) treat their operands as signed values in the range -32768 to 32767. The other operators and functions can be used within a signed integer expression, but still treat their operands as unsigned.

Based on how two's complement works, theoretically you can work with just the two's complement numbers (which if regarded as unsigned integers, are also positive integers, but in these cases that would be very cumbersome to have to remember the equivalents instead of the actual number we want to involve in our calculation).

I say we need to do $1 + (-32300)$ (1, what would be easier to implement?

```
PRINT % SGN {1} + SGN {-32300} - SGN {-1}
```

or

```
PRINT %1+33235 65535
```

There are obvious benefits on usability, and also non-obvious benefits such as in the following example:

```
10 XX=0
20 PRINT % (X-1)>0,
   %SGN{X-1}>0}
```

which will result in

```
1      0
```

on screen as in unsigned expressions 0-1 equals 65535 (see also the previous example) which is obviously larger than 0 while in signed expressions 0-1 equals -1 which is not larger than 0!

SGN { } also affects multiplication, division and MODulo operations. Consider this example (which also contains a pitfall!)

```
10 PRINT %10* 1
20 PRINT %10* SGN {1}
30 PRINT % SGN {10}* SGN {-1}}
40 PRINT % SGN {10}* 1}
```

If you RUN this, you will see the following on screen:

```
65526
65526
```



```
10
10
```

What happened here is that `-1` is as we discussed `65535` for unsigned integers. So on line 10, the computer multiplied `10 * 65535` which resulted in `655350` but as an integer number this is larger than 16 bits. Then it gets truncated to 16 bits which results into `65526` which is obviously wrong as a result! Moving to line 20 we hit the first pitfall discussed in the opening statement. The result of `SGN (-1)` which is `-1` gets converted into an unsigned integer itself so you end up with the exact same situation as with line 10: a multiplication of `10` with `65535`. The pitfall therefore here is that `SGN(x)` must apply to the entirety of the integer expression, so if there are other non-signed expressions they must be taken into consideration when writing each statement. Line 30 produces finally what we were aiming for, but that also happens with line 40. So both are correct but which is the right way to do it?

The answer to that question lies with what we discussed above regarding the "pitfall" with integer expressions. The subexpression `SGN (-1)` will get evaluated to what above is in the enclosing expression. So if the enclosing expression is an *unsigned* expression, the result of the subexpression will also become converted to *unsigned*, ergo since the entirety of the integer expression of line 30 is a signed expression, the signed subexpression is unnecessary and may even delay execution, especially in very complex calculations. The right way therefore to do it is the way defined in line 40. Obviously this also applied to our initial example which is best written as

```
PRINT X SGN (-1) 32380 (-1)
```

which is much neater to write AND read!

Exercises

- Using the discussion about the unary operator and 16 bit binary numbers, calculate and print on screen the *two's complement* for the signed 32 bit integer `650323`.

Chapter 7 Strings

Introduction

As we discussed previously strings are series of characters stored as numbers in memory. Which number represents which character is usually governed by a standard. For *NexBASIC* this standard is a modified version of the ASCII standard. In addition apart from characters, also defines tokens and UDGs. See Chapters 13 and 23 as well as *Appendix A* in order to better understand how characters, tokens and UDGs are stored. This chapter deals with ways of manipulating strings and reiterates some functions touched upon by previous chapters.

String slicing, using TO

Given a string, a substring of it consists of some consecutive characters from it, taken in sequence. Thus, string is a substring of "bigger string", but "b string" and "big reg" are not.

There is a notation called *slicing* for describing substrings, and this can be applied to arbitrary string expressions. The general form is

string expression (start TO finish)

so that, for instance

```
"abcdef"(2 TO 5) = "bcde"
```

if you omit the start, then 1 is assumed; if you omit the finish then the length of the string is assumed. Thus

```
"abcdef"( TO 5) = "abcdef"(1 TO 5) = "abcde"
```

```
"abcdef"(2 TO ) = "abcdef"(2 TO 6) = "bcdef"
```

```
"abcdef"( TO ) = "abcdef"(1 TO 6) = "abcdef"
```

(You can also write this last one as `"abcdef"()` for what it's worth)

A slightly different form misses out the TO and just has one number

```
"abcdef"(3) = "abcdef"(3 TO 3) = "c"
```

Although normally both start and finish must refer to existing parts of the string, this rule is overridden by another one: if the start is more than the finish, then the result is the empty string. So

```
"abcdef"(5 TO 7)
```

gives error **3 Subscript wrong** because the string only contains 6 characters and 7 is too many, but

```
"abcdef"(8 TO 7) = "" (an empty string)
```

and

```
"abcdef"(1 TO 0) = "" (again, an empty string)
```

The start and finish must not be negative, or you get error **B integer out of range**. This next program is a simple one illustrating some of these rules.

```
10 A$ = "abcdef"
20 FOR n=1 TO 6
30   PRINT A$(n TO 6)
40 NEXT n
```



```
50 STOP
```

Type **NEW** when this program has been run and enter the next program

```
10 a$='ABLE WAS I'
20 FOR n=1 TO 10
30 PRINT a$(n TO
    10),a$:(11-n) TO 10
40 NEXT n
```

For string variables we can not only extract substrings but also assign to them. For instance type

```
a$="I am the ZX Spectrum Next"
```

and then

```
a$(5 TO 8)="*****"
```

and

```
PRINT a$
```

Notice how since the substring `a$(5 TO 8)` is only 4 characters long only the first four stars have been used. This is a characteristic of assigning to substrings: the substring has to be exactly the same length afterwards as it was before. To make sure this happens, the string that is being assigned to it is cut off on the right if it is too long or filled out with spaces if it is too short. This is called 'Procrustean' assignment after the road bandit 'Procrustes' who used to make sure that his victims fitted the bed by either stretching them out on a rack or cutting their feet off.

If you now try

```
a$()=Hello there
```

and

```
a$," "
```

You will see that the same thing has happened again: this time with spaces put in because `a$()` counts as a substring.

```
a$=Hello there
```

will do it properly

Complicated string expressions will need parentheses around them before they can be sliced. For example

```
'abc'+ 'def'(1 TO 2)='abcde'
```

```
('abc'+ 'def')(1 TO 2)='ab'
```

String multiplication using the * operator

We saw earlier that the multiply (*) operator can be used for string replication. Its syntax is `a$*n` (takes a string `a$` as the first operand and a number `n` as the second operand, returning a string). If `n` has a fractional part then the string is replicated up to the character that represents the closest integer number to the product of `n` times the length of the original string. In other words, if you multiply a 7 character string by 1.5 times you will get the whole string and an additional 3 characters of it since $7 * 1.5 = 10.5$. Finally, the sign of the number determines whether the result is mirrored or not. Consider the following examples

<code>abcde fg '*2</code>	returns <code>abode fgabode f g</code> (two times the string)
<code>abcde fg '* 1</code>	returns <code>gfedc ba</code> (the string is inversed)
<code>'abcde fg '*1.5</code>	returns <code>abode f gabc</code>

Transforming a string with trailing modifiers

you follow a string expression `a$` by brackets containing a modifier list in the format `a$ (modifierlist)`

you can transform the string in several useful ways. The *modifierlist* can be any of the following characters:

<code>+</code>	convert lower case letters to upper case
	convert upper case letters to lower case
<code><</code>	strip leading spaces and control characters.
<code>></code>	strip trailing spaces (and control characters).
	strip bit 7 (More about bit 7 in Chapter 23) terminator from last character of string
<code>^</code>	add bit 7 terminator to last character of string
<code>(f\$,r\$)</code>	replace any occurrences of characters present in <code>f\$</code> with the corresponding character from <code>r\$</code> (or delete if there is no corresponding character)

The order of the modifiers is unimportant, except for `(f$,r$)` which, if present, must be the final modifier.

The following examples demonstrate what can be achieved:

```
'      Hello There      ' (<+ >)
```

gives `HELLO THERE`

as trailing and leading spaces are stripped via `<` and `>` and all lower case letters are converted to upper case with the `+` and upper case to lower case with the

```
'My typewriter is broken' (('nore', 'dro'))
```

returns `My typwrt is borkd` as `r$` doesn't have a corresponding letter for the `e` of `f$` so all of them are deleted, all `o` are replaced by `r` and all `r` are replaced by `o`

Note that it is possible to slice a modified string, or modify a sliced string, since both `()` and `[]` continue to be evaluated following a string argument, until there are no further opening parentheses or brackets. For example

```
a$ 5: [ ] (3 TO 7 [ < )
```

is perfectly valid

Tokenisation of strings

If need be (for example to easily prepare strings to be passed to functions `VAL` or `VAL $`. See next Chapter) we can tokenise, that is, convert NextBASIC reserved words to their single code equivalents (tokens, without syntax checking- the contents of a string expression `a$` by enclosing it in braces `{}` like so

```
{a$}
```

For example

<code>{ SIN (PI/4) }</code>	returns a string <code>"SIN (PI/4)"</code> including the tokens <code>SIN</code> (code 178) and <code>PI</code> (code 167)
-----------------------------	--

<code>VAL { 'SIN (PI/4) ' }</code>	gives 0.7071
------------------------------------	--------------

Chapter 8 Functions

Consider a sausage machine. You feed it meat in at one end, turn a handle and out comes a sausage at the other end. Providing pork meat gives us a pork sausage, beef a beef sausage and so on.

Functions are practically indistinguishable from sausage machines but there is a difference: they work on data instead of meat. You supply one value (called the argument) which it up by doing some calculations or transformations on it, and eventually get another value: the result.

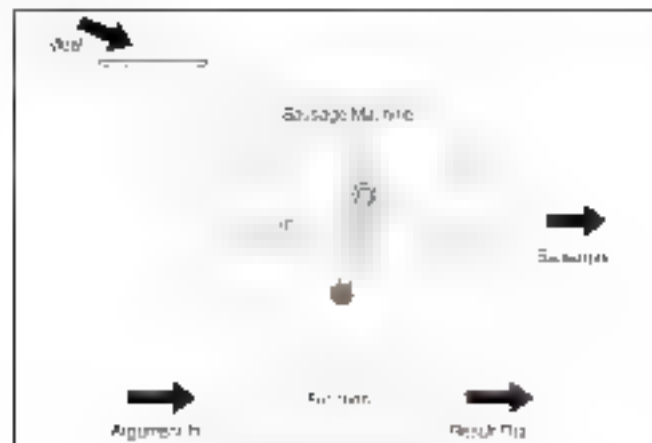


Fig. 8: How functions work

Different arguments give different results, and if the argument is completely inappropriate the function will stop and give an error report.

Just as you can have different machines to make different products – one for sausages, another for dish cloths, and a third for fish-fingers and so on, different functions will do different calculations. Each will have its own value to distinguish it from the others.

You use a function in expressions by typing its name followed by the argument, and when the expression is evaluated the result of the function will be worked out.

Functions always produce a specific datatype as result and as such they are used in the same way as variables, of a certain datatype in expressions. In a string-producing function names end with a \$ sign just like regular strings in order to not confuse us. For example running the following (see below for more regarding VAL)

```
10 avalue$ = '10'  
20 b = 10 + VAL (a value$)  
30 PRINT b
```

will produce 20 which is obviously a number. As far as expressions and NextBASIC are concerned VAL (x\$) can be used in lieu of any numeric variable or indeed number.

Order of calculations using functions

Expanding on what we saw on Chapter 6: if you mix functions and calculations in a single expression, then the functions' resulting values will be calculated first and before the rest of the operations. Again, however, you can circumvent this rule by using parentheses.

For instance, in the example below, here are two expressions which differ only in the parentheses, and yet the calculations are performed in an entirely different order in each case (although, as it happens, the end results are the same).

Each column shows the succession of operations according to their initial syntax which is found at the top of each column.

<code>LEN "Fred" + LEN "Bloggs"</code>	<code>LEN ("Fred" + "Bloggs")</code>
<code>4 + LEN "Bloggs"</code>	<code>LEN ("FredBloggs")</code>
<code>4 + 6</code>	<code>LEN "FredBloggs"</code>
<code>10</code>	<code>10</code>

Functions can be separated in *built in* (contained within *NexBASIC*) and *user defined* ones (the ones we write). Below we'll visit the *in-built* functions categorised according to the datatype we give them as input to manipulate and/or transform.

Please note that as the functions devoted to maths are a bit more complicated we will list them separately in their own chapter.

String functions

LEN

`LEN x$` works out the length of a string. Its single argument `x$` is the string whose length you want to find, and its result is the length, so that if you type

```
PRINT LEN "ZX Spectrum Next"
```

the answer 16 will be printed on screen, that is the number of characters in *ZX Spectrum Next* (spaces are counted as a character).

STR\$

`STR$` converts numbers into strings, that however is not its only capability and for that reason it has two forms. Apart from the simplest task of making a string out of a number in the way it would appear on screen displayed by a `PRINT` statement (that also means converted in case into a decimal number), it can also do the same but converting at the same time the number to a different base.

The simplest form `STR$x` doesn't use parentheses to contain its single argument `x` (a number), and its result is the string that would appear on the screen if the number were displayed by a `PRINT` statement. Note how its name ends in a `$` sign to show that its result is a string. For example, you could say

```
a$=STR$ 1e2
```

which would have exactly the same effect as typing

```
a$= 100
```

since the numeric argument gets converted from scientific notation to a standard decimal prior to printing. Or you could say

```
PRINT LEN STR$ 100 000
```

and get the answer 3, because `STR$ 100 000="100"`.

`STR$(x, base, places)` on the other hand uses parentheses, but takes up to 3 numeric arguments. It returns a string representation of number `x` in the optionally specified base (2 to 36). For bases > 10, digits larger than 9 are represented with capital letters starting with `A`. If the base is not specified it's assumed it's 10. If optional argument `places` is present, then a fractional part of places digits is also output. Let's first rewrite the example above in a couple of ways

```
a$=STR$ (1e2,
a$=STR$ (1e2, 16
```

If you assumed prior to trying that the first produces the same result as the `STR$` without parentheses (further above) you'd be correct: it too returns 100. The second however demonstrates the power of conversion as we told `STR$` to convert to base 16 (hexadecimal). This returns 64 which is 100 expressed in the hexadecimal system. The following two

examples show what happens when we request both a conversion and the optional places

```
STR$ 201.5, 16
```

which gives us C9 which equals 20¹ the integer part of 201.5 and

```
STR$ 3.5, 2, 4
```

which converts 3.5 into a binary number with 4 fractional places for .5 and returns 11.1000

IN

IN (source\$, match\$, startpos, wild\$,) returns the leftmost character position number where match\$ was found in source\$. Optional parameter startpos determines the position within source\$ to begin the search (default = 1). If startpos is negative, the search starts from position **ABS**(startpos) and proceeds backwards. (More about **ABS** further below)

The value returned will be between 1 and **LEN** source\$. If a match was found or 0 if a match is not found, startpos is 0, match\$ or source\$ are the empty string ("").

If you're not entirely sure of the spelling of the word you're looking for within the original string you have the option to use a wildcard character which itself is user-definable. Any character you enter in wild\$ or the copyright symbol © (ASCII 127) will become the wildcard character which you can then substitute in match\$ for any character you're unsure about. Any characters in match\$ which are the wildcard character will match any character in source\$. If wild\$ is the empty string the wildcard character is ASCII 0. Here are some examples to better illustrate

```
PRINT IN("Harry is awesome", " ")
```

Prints 6 on the screen as a space is a valid character while

```
PRINT IN("Harry is awesome",
, 7)
```

Prints 9 as we told IN to start looking from position 7 onwards

```
PRINT IN("Harry is awesome", "
", 16)
```

will also return 9 as it's looking backwards and finally

```
PRINT IN("Here's
Garry! a%y 1 % ")
```

prints 9 as we substituted any letter for the symbol % and the first string is located there. If however we modified the above slightly and made it

```
PRINT IN("Here's
Garry!" "%", 1, "%")
```

we'd get 2 as the first match for the %r is located there. If we then modified it a little further to

```
PRINT IN("Here's
Garry!" "%r", 5, "%")
```

it would have returned 10 as starting from position 5 the first match is the rr in Garry

A wildcard character is a character placeholder for any possible character

VAL and VAL\$

VAL *x\$* is like **STR\$** in reverse as it converts string *x\$* into the numeric representation of said string. For instance

```
VAL '3.5'
```

returns 3.5. That is because if you take any number, apply **STR\$** to it, and then apply **VAL** to it, you will get back to the number you first thought of. That being said, if you take a string, apply **VAL** to it, and then apply **STR\$** to it, you do not always get back to your original string.

VAL is an extremely powerful function, because the string which is its argument is not restricted to looking like a plain number. It can be any numeric expression. Thus, for instance

```
VAL '2*3'
```

will return 6 or even

```
VAL ('2' + '*3')
```

will return the same. There are two things happening here. In the first, the argument of **VAL** is evaluated as a string, the string expression "2*+*3" is evaluated to give the string "2*3". Then, the string has its double quotes stripped off, and what is left is evaluated as a number, so 2*3 is evaluated to give the number 6.

Now, the following can get pretty confusing pretty fast if you do not pay the requisite attention. Remember that inside a string a string quote must be written twice. If you go down into further depths of strings, then you find that string quotes need to be quadrupled or even octupled.

There is another function, rather similar to **VAL**, called **VAL\$**. Its argument *x\$* is still a string, but its result is also a string. To see how this works, recall how **VAL** functions in two steps: first, its argument is evaluated as a string, then the double quotes are stripped off this, and whatever is left is evaluated as a number. With **VAL\$**, the first step is the same, but after the string quotes have been stripped off in the second step, whatever is left is evaluated as another string. Thus

```
VAL$ ""'Fruit punch'"
```

equals to Fruit Punch (Notice how the string quotes proliferate again. Do

```
a$="99"
```

and print out all of the following: **VAL a\$**, **VAL 'a\$'**, **VAL ""a\$""**, **VAL\$ a\$**, **VAL\$ "a\$"** and **VAL\$ ""a\$""**. Some of these will work, and some of them won't. Try to explain all the answers. (Try not to get overly confused.)

Numeric functions

SGN

SGN *x* is the sign function (sometimes called signum). It is the first function you have seen that has nothing to do with strings, because both its argument *x* and its result are numbers. The result is +1 if the argument is positive, 0 if the argument is zero, and -1 if the argument is negative.

ABS

ABS x is another function whose argument x and result are both numbers. It converts the argument into a positive number which is the result, by stripping the sign away, so that for instance

```
ABS 3.2
```

is the same as

```
ABS 3.2
```

which in turn equals 3.2

INT

INT x stands for *integer part*. An integer is a whole number, possibly negative. This function converts a fractional number x into an integer by throwing away the fractional part, so that for instance

```
INT 3.9
```

equals 3. Be careful when you are applying it to negative numbers, because it always rounds down, thus, for instance

```
INT -3.9
```

will return -4

SQR

SQR calculates the square root of a number. The result is that which multiplied by itself gives the argument. For instance

```
SQR 4
```

returns 2 because $2*2 = 4$

```
SQR 0.25
```

returns 0.5 because $0.5*0.5 = 0.25$ and finally

```
SQR 2
```

will return 1.4142136 (approximately) because $1.4142136*1.4142136=2.0000001$

If you multiply any number, even a negative one, by itself, the answer is always positive. This means that negative numbers do not have square roots, so if you apply **SQR** to a negative argument you get an error: **An Invalid Argument**.

User defined functions using DEF and FN

You can also define functions of your own. Their names follow exactly what is valid for procedures. Their arguments/parameters follow the conventions about procedures as well. Additionally, just like procedures and subroutines, functions can also be recursive. Recall the factorial example from Chapter 4 and let's try to express it in a function form.

```
DEF FN factor (n) =n*(1,1,n*FN
    factor (n-1))
```


You define a function by putting a DEF statement somewhere in the program. For instance, here is the definition of a function **FN sq** whose result is the square of the argument:

```
DEF FN sq(x)=x*x REM square of x
```

The **sq** following the DEF FN is the name of the function. The **x** in parentheses is a name by which you wish to refer to the argument of the function.

After the = sign comes the actual definition of the function. This can be any expression and it can also refer to the argument using the name you've given it (in this case **x**) as though it were an ordinary variable.

When you have entered this line, you can invoke the function just like one of the computer's own functions, by typing its name **FN sq** followed by the argument. Remember that when you have defined a function yourself, the argument must be enclosed in parentheses. Try it out a few times:

```
PRINT FN sq(2)
PRINT FN sq(3+4)
PRINT 1+INT FN sq (LEN 'chicken')/2+3,
```

Once you have put the corresponding DEF statement into the program, you can use your own functions in expressions just as freely as you can use the computer's.

Note: in some dialects of BASIC you must even enclose the argument of one of the computer's functions in parentheses. This is not the case in *NextBASIC*.

INT always rounds down. To round to the nearest integer, add .5 first: you could write your own function to do this:

```
20 DEF FN r(x)=INT (x+.5)
REM gives x rounded to the
nearest integer.
```

You will then get, for instance:

```
FN r(2.9) = 3    FN r(2.4) = 2
FN r(-2.9) = -3  FN r(-2.4) = -2
```

Compare these with the answers you get when you use **INT** instead of **FN r**. Type in and run the following:

```
10 x,y,a=0,0,10
20 DEF FN p(x,y)=a+x*y
30 DEF FN q(x)=a+x*y
40 PRINT FN p(2,3),FN q()
```

There are a lot of subtle points in this program.

First, a function is not restricted to just one argument: it can have more, or even none at all, but you must still always keep the parentheses.

Second, it doesn't matter whereabouts in the program you put the DEF FN statements. After the computer has executed line 10 it simply skips over lines 20 and 30 to get to line 40. Key 10, however, have to be somewhere in the program: they can't be in a comment.

Third, **x** and **y** are both the names of variables in the program as a whole, and the names of arguments for the function **FN p**. **FN p** temporarily forgets about the variables called **x** and **y**, but since **f** has no argument called **a**, it still remembers the variable **a**. In other words the exposed parameters of a function are always **LOCAL** within the function.

Thus when **FN p(2,3)** is being evaluated, **a** has the value 10 because it has been initialised in line 11, **x** has the value 2 because it is the first argument, and **y** has the value 3 because it is the second argument. Both **x** and **y** albeit also defined in line 10 are treated as **LOCAL** variables and their value doesn't modify their global counterparts. The result is then $10+2*3=16$.

When **FN q()** is being evaluated, on the other hand, there are no arguments. So **a**, **x** and **y** all still refer to the variables and have values 10, 0 and 0 respectively. The answer in this case is $10+0*0=10$.

Now change line 20 to

```
20 DEF FN p(x,y)=FN q(
```

This time **FN p(2,3)** will have the value 10 because **FN q** will still go back to the variables **x** and **y** rather than using the arguments of **FN p**.

DEF FN can take parameters passed by **REFERENCE** as is obvious from the next example:

```
DEF FN jenz$(REF
    jenz(1,1dx)=jenz(1dx)
```

Some BASICs (but not *NextBASIC*) have functions called **LEFT\$**, **RIGHT\$**, **MID\$** and **TL\$**.

LEFT\$ (a\$,n) gives the substring of **a\$** consisting of the first **n** characters.

RIGHT\$ (a\$,n) gives the substring of **a\$** consisting of the characters from **n**th on.

MID\$ (a\$,n₁,n₂) gives the substring of **a\$** consisting of **n₂** characters starting at the **n₁th**.

TL\$ (a\$) gives the substring of **a\$** consisting of all its characters except the first.

You can write some user-defined functions to do the same. For example

```
10 DEF FN TL$(a$)=a$ 2 TO 1
20 DEF FN LEFT$(a$,n)=a$(1 TO
    n)
```

Check that these work with strings of length 0 or 1.

Note that our **FN LEFT\$** has two arguments, one a number and the other a string.

A function cannot have integer arguments, nor use integer expressions in its definitions.

NextBASIC functions within integer expressions

We already discussed the usage of the **INT { }** keyword which converts any floating point expression into an integer expression, but in many cases this can be slow. In other cases the values produced by a function are anyway plain 8 or 16 bit integers which means that integer-only versions of said function would provide significant boost over their standard counterparts. *NextBASIC* caters for these cases with special integer-only forms of the following functions:

ABS n	Return the ABSolute value of n . See this Chapter
IN n	Read value from Hardware Port n . See Chapter 22
INPUT n	Read/Define input controllers. See Chapters 17 and 22
REG n	Read value from Next Register n . See Chapter 22
PEEK a	Read byte from address a in memory. See Chapter 23

DPEEK a	Read word ^a from memory (double PEEK) See Chapter 23
USR^b a	Execute Machine Code routine See Chapter 25
USR\$ a	Execute Machine Code routine See Chapter 25
BIN n	Synonym for @n specifying binary values
RND n	Generates pseudo-random value in range 0 to n-1 (equivalent to floating-point INT (RND*n))
BANK b PEEK o	Read byte at offset o from bank b See Chapter 23
BANK b DPEEK o	Read word at offset o from bank b (double PEEK) See Chapter 23
BANK b USR o	Execute Machine Code routine at offset o in bank b See Chapters 23 and 25
BANK b USR\$ o	Execute Machine Code routine at offset o in bank b See Chapters 23 and 25

These are written by including a % sign in front of them like all integer expressions. For example to read from hardware port 254:

```
%a = % IN 254
```

Or to check what speed your ZX Spectrum Next is running, masking the speed bits of NextREG 7 you could give:

```
PRINT %REG 7 & BIN 00000011
```

Randomly read a byte from the ROM:

```
%a=%RND 16384 PRINT %a,% PEEK a
```

Exercise

- 1 Use the function FN sq(x)=x*x to test SQR. You should find that

```
FN sq(SQR x)=x
```

if you substitute any positive number for x, and

```
SQR FN s(x)=ABS x
```

whether x is positive or negative (Why the ABS?)

- 2 Write functions FN RIGHTS and FN MID\$

^a A word in standard computer terminology is a two-byte (ie. 16 bit) value. In values (two-word) are called long words.

^b USR and USR\$ are very special functions as they also act like commands but using a function syntax.

Chapter 9 Mathematical Functions

This chapter deals with the mathematics that the ZX Spectrum Next can handle. Quite possibly you will never have to use any of this at all, so if you find it too heavy going, don't be at all afraid of skipping it. It covers the operation \uparrow (raising to a power), the functions **EXP** and **LN**, and the trigonometrical functions **SIN**, **COS**, **TAN** and their inverses **ASN**, **ACS** and **ATN**.

\uparrow and **EXP**

You can raise one number to the power of another, that means *multiply the first number by itself the second number of times*. This is normally shown by writing the second number just above and to the right of the first number like so 2^3 , but since this gets unnecessarily complex to write and display on a computer, we use the symbol \uparrow instead. For example the powers of 2 are

$$\begin{aligned}2\uparrow1 &= 2 \\2\uparrow2 &= 2*2 = 4 && \text{(2 squared)} \\2\uparrow3 &= 2*2*2 = 8 && \text{(2 cubed)} \\2\uparrow4 &= 2*2*2*2 = 16 && \text{(2 to the fourth power)}\end{aligned}$$

Thus at its most elementary level $a\uparrow b$ means *a multiplied by itself b times*, but obviously this only makes sense if b is a positive whole number. To find a definition that works for other values of b, we consider the rule

$$a\uparrow(b+c) = a\uparrow b * a\uparrow c$$

(Notice that we give \uparrow a higher priority than $*$ and so that when there are several operations in one expression, the \uparrow s are evaluated before the $*$ s and /s.) You should not need much convincing that this works when b and c are both positive whole numbers, but if we decide that we want it to work even when they are not, then we find ourselves compelled to accept that

$$\begin{aligned}a\uparrow0 &= 1 \\a\uparrow(-b) &= 1/a\uparrow b \\a\uparrow(1/b) &= \text{the } b\text{th root of } a, \text{ which is to say the number that you have to} \\&\text{multiply by itself } b \text{ times to get } a\end{aligned}$$

and

$$a\uparrow(b*c) = (a\uparrow b)\uparrow c$$

If you have never seen any of this before, then don't try to remember it straight away, just remember that

$$a\uparrow(-1) = 1/a$$

and

$$a\uparrow(1/2) = \text{SQRT } a$$

and maybe when you are familiar with these, the rest will begin to make sense.

Experiment with all this by trying this program

```
10 INPUT a, b, c
20 PRINT a^(b+c), a^b*a^c
30 GO TO 10
```

Of course, if the rule we gave earlier is true, then each time round the two numbers that the computer prints out will be equal. (Note, because of the way the computer works out \uparrow , the number on the left, a in this case, must never be negative.)

A rather typical example of what this function can be used for is that of compound interest. Suppose you keep some of your money in a building society and they give 5% interest per year. Then after one year you will have not just the 100% that you had anyway, but also the 5% interest that the building society have given you, making altogether 105% of what you had originally. To put it another way, you have multiplied your sum of money by 1.05, and this is true however much you had there in the first place. After another year the same will have happened again, so that you will then have $1.05 \times 1.05 = 1.05^2 = 1.1025$ times your original sum of money. In general, after y years you will have 1.05^y times what you started out with.

If you try this command

```
FOR y=0 TO 100 PRINT y,10*1.15^y
NEXT y
```

you will see that even starting off from just £10, it all mounts up quite quickly, and what is more, it gets faster and faster as time goes on. (Although even so, you might still find that it doesn't keep up with inflation.)

This sort of behaviour, where after a fixed interval of time some quantity multiplies itself by a fixed proportion, is called *exponential growth*, and it is calculated by raising a fixed number to the power of the time. Suppose you did this

```
10 DEF FN a(x)=a^x
```

Here a is more or less fixed by LET statements; its value will correspond to the interest rate, which changes only every so often.

There is a certain value for a that makes the function FN a look especially pretty to the trained eye of a mathematician and this value is called e . NextBASIC has a function called EXP defined by

```
EXP x=e^x
```

Interestingly, e itself is not an especially pretty number; it is an infinite non-recurring decimal. You can see its first few decimal places by doing

```
PRINT EXP 1
```

because $\text{EXP } 1 = e^1 = e$. Of course, this is just an approximation. You can never write down e exactly.

LN

The inverse of an exponential function is a logarithmic function. The *logarithm* 'to base a ' of a number x is the power to which you have to raise a to get the number x , and it is written $\log_a x$. Thus by definition $a^{\log_a x} = x$, and it is also true that $\log(a^x) = x$. You may well already know how to use *base-10 logarithms* for doing multiplications; these are called *common logarithms*. NextBASIC has a function LN which calculates *logarithms to the base e* ; these are called *natural logarithms*. To calculate logarithms to any other base, you must divide the *natural logarithm* by the *natural logarithm* of the base:

$$\log_a x = \text{LN } x / \text{LN } a$$

P

Given any circle, you can find its perimeter (the distance round its edge, often called its *circumference*) by multiplying its diameter (width) by a number called π . π is a Greek p , and it is used because π stands for the Greek word *perimeter*. Unlike what's commonly believed, its pronunciation is the same as in English.

Like e , π is an infinite non-recurring decimal. It starts off as 3.141592653589... The word **PI** in NextBASIC is taken as standing for this number. Try **PRINT PI**.

Trigonometry with SIN, COS, TAN, ASN, ACS and ATN

The trigonometrical functions measure what happens when a point moves round a circle. Here is a circle of radius 1.1 what? It does it all in after as long as we keep to the same unit all the way through. There is nothing to stop you inventing a new unit in your own, or every circle that you happen to be interested in, and a point moving round it. The point started at the 3 o'clock position, and then moved round in an anti-clockwise direction.

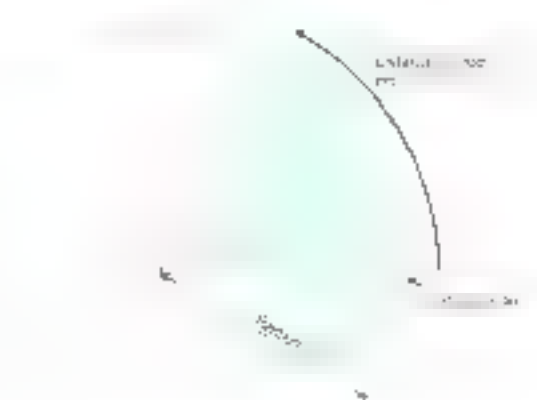


Fig. 6 Basis of trigonometrical measurements

We have also drawn in two lines called axes through the centre of the circle. The one through 9 o'clock and 3 o'clock is called the *x*-axis, and the one through 6 o'clock and 12 o'clock is called the *y*-axis. To specify where the point is, you say how far it has moved round the circle from its 3 o'clock starting position: let us call this distance *a*. We know that the circumference of the circle is 2π (because its radius is 1, and its diameter is thus 2), so when it has moved a quarter of the way round the circle, $a = \pi/2$, when it has moved halfway round, $a = \pi$, and when it has moved the whole way round, $a = 2\pi$.

Given the curved distance round the edge, *a*, two other distances you might like to know are how far the point is to the right of the *y*-axis, and how far it is above the *x*-axis. These are called, respectively, the *cosine* and *sine* of *a*. The functions **COS** and **SIN** on the computer will calculate these.

Note that if the point goes to the left of the *y*-axis, then the *cosine* becomes negative, and if the point goes below the *x*-axis, the *sine* becomes negative.

Another property is that, once *a* has got up to 2π , the point is back where it started and the *sine* and *cosine* start taking the same values all over again.

$$\text{SIN}(a + 2\pi) = \text{SIN } a$$

$$\text{COS}(a + 2\pi) = \text{COS } a$$

The *tangent* of *a* is defined to be the *sine* divided by the *cosine*; the corresponding function on the computer is called **TAN**.

Sometimes we need to work these functions out in reverse, finding the value of *a* that has given *sine*, *cosine* or *tangent*. The functions to do this are called *arcsine* (**ASN** on the computer), *arccosine* (**ACS**) and *arctangent* (**ATN**).

In the diagram of the point moving round the circle look at the radius joining the centre to the point. You should be able to see that the distance we have called a (the distance



Fig. 7 Graphical representation of trigonometrical functions

that the point has moved round the edge of the circle is a way of measuring the angle through which the radius has moved away from the x-axis.

When $a = \pi/2$, the angle is 90° (degrees)

When $a = \pi$ the angle is 180° and so round to when $a = 2\pi$ and the angle is 360°

You might just as well forget about degrees and measure the angle in terms of a alone: we say then that we are measuring the angle in radians. Thus $\pi = 2$ radians, 90° and so on.

You must always remember that in NextBASIC **SIN**, **COS** and so on use radians and not degrees. To convert degrees to radians, divide by **180** and multiply by π . To convert back from radians to degrees, you divide by π and multiply by **180**.

Exercises

Using the knowledge you have gained from this chapter, define a function to convert radians to degrees (this may prove very useful to you in the future)

2. In Fig. 7 above, the function **COT** appears while it's not part of NextBASIC's vocabulary. Write a function that returns the value of the cotangent of a using **TAN**.

Chapter 10 Random Numbers

RANDOMIZE RND and %RND

This chapter deals with the functions **RND**, **RND()** and **%RND** and the keyword **RANDOMIZE**. They are all used in connection with random numbers, so you must be careful not to get them mixed up.

As far as normal functions go, **RND** is quite unusual, although it does calculations and produces a result, it does not need an argument.

Each time you use it, its result is a new random floating-point number between 0 and 1. (Sometimes it can take the value 0, but never 1.)

Try

```
10 PRINT RND
20 GO TO 10
```

to see how the answer varies. Can you detect any pattern? You shouldn't be able to: random means that there is no pattern.

%RND, which is, as seen on Chapter 8, the version of **RND** available in integer expressions, behaves slightly differently: it takes a single argument, i.e. *n*, and returns a random integer in the range 0 to *n*-1. For example, **%RND 10** will return a random integer between 0 and 9.

While **RND** returns, as discussed above, a random number between 0 and 1, you can easily get random numbers in other ranges. For instance, **5*RND** is between 0 and 5, and **3+0.7*RND** is between 1.3 and 2. For cases where we need to use in the standard expression evaluator, there is yet another version of **RND**, which is not an integer expression only function.

RND(n)

which returns a random integer between 0 and *n*-1 like **%RND n**. This is recommended over using the standard fractional floating-point function **RND** since it doesn't suffer from the biasing inherent in converting a fractional random number to an integer with multiply and truncation steps.

To get whole numbers with **RND** use **INT**, remembering that **INT** always rounds down, as in **1+INT(RND*6)**—however your desired random values can stay within the range of 0 to 65535, it is better to use **%RND** or **RND()** which avoid the unnecessary conversions—and rather slow—floating point calculations involved.

To illustrate better what all version can do, let's use all three of them in a program to simulate dice throwing. **RND*6** is in the range 0 to 6, but since it never actually reaches 6, **INT(RND*6)** is 0, 1, 2, 3, 4 or 5.

Here is the dice throwing program:

```
10 REM dice throwing program
20 CLS
30 FOR n=1 TO 2
40 PRINT 1+INT(RND*6), " ",
50 NEXT n
60 INPUT a$ GO TO 20
```

* Actually, **RND** is not truly random, because it follows a fixed sequence of 65536 numbers. However, these are so thoroughly jumbled up that there are at least no obvious patterns so we say that **RND** is pseudo-random.

Press **ENTER** each time you want to throw the dice. To use **% RND** instead change line 40 to read

```
40 PRINT %1+ RND 6, " ",
```

and to use **RND()** you need to write line 40 as

```
40 PRINT 1+ RND (6), " ",
```

Aren't the latter two more readable?

The **RANDOMIZE** statement is used to make **RND** and **% RND** start off at a definite place in its sequence of numbers as you can see with this program

```
10 RANDOMIZE 1
20 FOR n=1 TO 5 PRINT % RND
  100, NEXT n
30 PRINT GO TO 10
```

After each execution of **RANDOMIZE 1** the **% RND** sequence starts off again with 97 and if you use **RND** instead of **% RND 100** you'll get 0 0022735596. You can use other numbers between 1 and 65535 in the **RANDOMIZE** statement to start the **RND** sequence off at different places.

If you had a program with **RND**, **RND()** or **%RND** in it and it also had some mistakes that you had not found, then it would help to use **RANDOMIZE** like this so that the program behaved the same way each time you ran it.

RANDOMIZE on its own and **RANDOMIZE 0** has the same effect, is different because it really does randomise **RND**, **RND()** and **% RND** you can see this in the next program.

```
10 RANDOMIZE
20 PRINT % RND 65535 GO TO 10
```

The sequence you get here is not very random, because **RANDOMIZE** uses the time since the computer was switched on. Since this has gone up by the same amount each time **RANDOMIZE** is executed, the next **% RND** does more or less the same. You would get better randomness by replacing **GO TO 10** by **GO TO 20**. Here is a program to toss coins and count the numbers of heads and tails.

```
10 heads,tails=0
20 coin=% RND 2
30 ON coin heads+=1 tails+=1
40 PRINT heads," ",tails,
50 IF tails<>0 THEN PRINT
  heads/tails,
60 PRINT GO TO 20
```

The ratio of heads to tails should become approximately 1 if you go on long enough because in the long run you expect approximately equal numbers of heads and tails.

Note that **RANDOMIZE** can also be written in short as **RAND** and it will expand to **RANDOMIZE**.

Exercises

1 (For mathematicians only)

Let p be a (large, prime) and let a be a primitive root *modulo* p .

Then b_i is the residue of a *modulo* p ($1 \leq b_i \leq p-1$) the sequence

$$\{b_i\}_{i=1}^{p-1}$$

is a cyclical sequence of $p-1$ distinct numbers in the range 0 to 1 (excluding 1).

By choosing a suitably these can be made to look fairly random.

65537 is a Fermat prime $2^{16}+1$ because the multiplicative group of non-zero residues *modulo* 65537 has a power of 2 as its order. A residue is a primitive root if and only if it is not a quadratic residue. Use Gauss' law of quadratic reciprocity to show that 75 is a primitive root *modulo* 65537.

The ZX Spectrum Next uses $p=65537$ and $a=75$ and stores some b_i in memory. RND entails replacing b_i in memory by $b_i + 1$ and yielding the result $\{b_i\}_{i=1}^{p-1}$.

RANDOMIZE n (with $1 \leq n \leq 65535$) makes b_i equal to $n+1$.

RND is approximately uniformly distributed over the range 0 to 1.

Chapter 11 Arrays

DIM

Suppose you have a list of numbers, for instance the marks of ten people in a class. To store them in the computer you could set up a single variable for each person, but you would find them very awkward. You might decide to call the variable `Bloggs 1`, `Bloggs 2`, and so on up to `Bloggs 10`, but the program to set up these ten numbers would be rather long and boring to type in:

How much nicer it would be if you could type this:

```
5 REM this program will not
  work
10 FOR n=1 TO 10
20   READ B o g g s n
30 NEXT n
40 DATA 10,2,5,9,16,3,11,1,0,6
```

Well, you can't

However, there is a mechanism by which you can apply this idea, and it uses arrays. An array is a set of variables, its *elements*, all with the same name, and distinguished only by a number (the *subscript*) written in parentheses after the name. In our example the name could be `b` and the ten variables would then be `b(1)`, `b(2)`, and so on up to `b(10)`.

The *elements* of an array are called *subscripted variables*, as opposed to the simple variables that you are already familiar with.

Before you can use an array, you must reserve some space for it inside the computer, and you do this using a **DIM** (for dimension) statement:

```
DIM b(10)
```

sets up an array called `b` with dimension 10 (i.e. there are 10 subscripted variables `b(1)` to `b(10)`) and initialises the 10 values to 0. It also deletes any array called `b` that existed previously. (But not a simple variable. An array and a simple numerical variable with the same name can coexist, and there shouldn't be any confusion between them because the array variable always has a *subscript*.) The subscript can be an arbitrary numerical expression, so now you can write

```
5 DIM b(10)
10 FOR n=1 TO 10
20   READ b(n)
30 NEXT n
40 DATA 10,2,5,9,16,3,11,1,0,6
```

to read in the elements from a **DATA** list, or

```
10 FOR %n=1 TO 10
20 INPUT %m(n)
30 NEXT %n
```

to **INPUT** the elements' values by hand. Note, that in the second example there is no **DIM** statement. That's because as discussed in Chapter 1, the second array is an *integer array*. Integer arrays come predimensioned to a fixed 64 elements numbered 0 to 63. Attempting

to enter a DIM statement for %m will produce an audible tone and entering the statement will not be successful.

If we need to use an integer array with more than 64 elements, it is possible although what changes is the way we have to address them. Whereas in a normal integer array the subscript is written inside parentheses (), for integer arrays larger than 64 elements the subscript is written within brackets []. Furthermore, larger-than-64-elements integer arrays reduce the number of available integer arrays in the system as they take the entire array that follows sequentially from the one we're using and attach it to the current one. What this means is that if we want to use a 128-element integer array %a, this will take the space from integer array %b(). If we want to use an 192-element integer array %c, this will use space from integer arrays %d() and %e() and so on.

The maximum integer array usable is $26 \times 64 = 1664$ if using integer array %a[] with no other arrays available. Note that subsequent arrays don't disappear; they're still accessible carrying data from the integer array that reserved them. Modifying them however may have unexpected consequences. Thus at this point let's assume an integer array %a, with a desired 128 elements. Write the following little program:

```
10  A[65] = 43
20  PRINT %a[65]
30  PRINT %b[1] REM the 65th
    element of array a[] is
    b[1]
```

It's now obvious how this works.

You can also set up arrays with more than one dimension. This does also apply to Integer Arrays although they're normally predefined to have a single dimension; you'll see how below. In a two-dimensional array you need two numbers to specify one of the elements, rather like the line and column numbers to specify a character position on the television screen, so it has the form of a table or matrix.

Alternatively, if you imagine the line and column numbers, two dimensions, as referring to a printed page, you could have an extra dimension for the page numbers. Of course we are talking about numeric arrays, so the elements would not be printed characters as in a book, but numbers. Think of the elements of a three-dimensional array *v* as being specified by *v* (page number, line number, column number).

For example, to set up a two-dimensional array *c* with dimensions 3 and 6, you use a DIM statement:

```
DIM c(3,6)
```

This then gives you $3 \times 6 = 18$ subscripted variables:

	1	2	3	4	5	6
1	c(1,1)	c(1,2)	c(1,3)	c(1,4)	c(1,5)	c(1,6)
2	c(2,1)	c(2,2)	c(2,3)	c(2,4)	c(2,5)	c(2,6)
3	c(3,1)	c(3,2)	c(3,3)	c(3,4)	c(3,5)	c(3,6)

Table 1 Representation of a two-dimensional array

The same principle works for any number of dimensions.

Although you can have a number and an array with the same name, you cannot have two arrays with the same name, even if they have different numbers of dimensions except in the case of normal numerical and integer arrays.

As we mentioned above integer arrays can have a second dimension as well. This follows the discussion of extending integer arrays to larger than 64 elements. The technique is similar. If a two-dimensional integer array is required, we enclose subscripts within brackets. The difference here is that subscripts need to be individually enclosed. For example, whereas we would address regular array `c()` defined with `DIM c(4,64)` with `c(x,y)` in the case of its integer counterpart, we would address it as `%c(x,y)`. Each `x` denotes or takes one entire array that follows the base array name. For example, using `%c(x,y)` with `x=0` to 5 and `y=0` to 63 will use arrays `%C()`, `%D()`, `%E()`, `%F()`, `%G()` and `%H()`.

There are also *string arrays*. The strings in an array differ from simple strings in that they are of fixed length and assignment to them is always Procrustean—chopped off or padded with spaces. Another way of thinking of them is as arrays (with one extra dimension) of single characters. The name of a string array is a standard variable name followed by `$` and a string array and a simple string variable cannot have the same name (unlike the case for numbers).

Suppose then that you want an array `a$` of three strings. You must decide how long these strings are to be. Let us suppose that 10 characters each is long enough. You then say

```
DIM a$(3,10)      (type this in)
```

This sets up a 3*10 array of characters, but you can also think of each row as being a string.

		1	2	3	4	5	6	7	8	9	10
1	a\$(1)	a\$(1,1)	a\$(1,2)	a\$(1,3)	a\$(1,4)	a\$(1,5)	a\$(1,6)	a\$(1,7)	a\$(1,8)	a\$(1,9)	a\$(1,10)
2	a\$(2)	a\$(2,1)	a\$(2,2)	a\$(2,3)	a\$(2,4)	a\$(2,5)	a\$(2,6)	a\$(2,7)	a\$(2,8)	a\$(2,9)	a\$(2,10)
3	a\$(3)	a\$(3,1)	a\$(3,2)	a\$(3,3)	a\$(3,4)	a\$(3,5)	a\$(3,6)	a\$(3,7)	a\$(3,8)	a\$(3,9)	a\$(3,10)

Table 4. Representation of a string array

If you give the same number of subscripts (two in this case) as there were dimensions in the `DIM` statement, then you get a single character, but if you miss the last one out, then you get a *fixed-length string*. So, for instance, `a$(2,7)` is the 7th character in the string `a$(2)`. Using the slicing notation, we could also write this as `a$(2)(7)`. Now type

```
a$(2) = '1234567890'
```

and

```
PRINT a$(2),a$(2,7)
```

You get

```
1234567890      7
```

For the last subscript (the one you can miss out) you can also have a slice, so that for instance

```
a$(2,4 TO 8) = a$(2)(4 TO 8) = '45678'
```

Remember, in a string array, all the strings have the same *fixed* length. The `DIM` statement has an extra number (the last one) to specify this length. When you write down a subscripted variable for a string array, you can put in an extra number (or a slice) to correspond with the extra number in the `DIM` statement. You can have string arrays with no dimensions. Type

```
DIM a$(10,
```

and you will find that `a$` behaves just like a string variable, except that it always has length 10, and assignment to it is always Procrustean. Whatever part of the value doesn't fit gets left out.

DIM function

Apart from the **DIM** array declaration command, there's also a function by the same name that returns information regarding any declared array. Its syntax is

DIM (arrayname \$(f) [dimension]) : numeric and it returns the number of elements in the specified dimension of the array arrayname (dimension defaults to 0 if not specified)

If dimension equals 0, **DIM** will simply return the number of dimensions in the array

Simple strings are treated as single-dimension character arrays, returning 1 as the number of dimensions and the current string length as the number of elements in dimension 1. Let's write a little program to illustrate

```
10 DIM a (100,10,5)
20 PRINT DIM (a())
30 PRINT DIM(a(),1)
40 PRINT DIM(a(),2)
50 PRINT DIM(a(),3)
```

which will return

```
3
100
10
5
```

Exercises

- 1 Use **READ** and **DATA** statements to set up an array **m\$** of twelve strings in which **m\$(n)** is the name of the *n*th month. Hint: the **DIM** statement will be **DIM m\$(12,9)**. Test it by printing out all the **m\$(n)** (use a loop).

- 2 Type

```
PRINT "now is the month of
",m$(5),"ing", " when
merry lads
are playing"
```

What can you do about all those spaces?

Chapter 12 Conditions

AND, OR and NOT

We saw in Chapter 2 how an **IF** statement takes the form

IF condition

Apart from expressions that generate true (1) or false (0), res. lls, the conditions here were the relations (= < > <= >= and <>), which compare two numbers or two strings. You can also combine several of these using the logical operations, **AND**, **OR** and **NOT**.

One relation **AND** another relation is true whenever both relations are true, so you could have a line like

```
IF a$='yes' AND x>0 THEN PRINT x
```

in which **x** only gets printed if **a\$='yes'** and **x>0**. The syntax here is so close to English that it hardly seems worth spelling out the details. As in English, you can join lots of relations together with **AND** and then the whole lot is true if all the individual relations are.

One relation **OR** another is true whenever at least one of the two relations is true. (Remember that it is still true if both the relations are true: this is not always implied in English.)

The **NOT** relationship turns things upside down. The **NOT** relation is true whenever the relation is false, and false whenever it is true.

Logical expressions can be made with relations and **AND**, **OR** and **NOT** just as numerical expressions can be made with numbers and + and so on: you can even put them in parentheses if necessary. They have priorities in the same way as the usual operations: +

* and ^ do. **OR** has the lowest priority, then **AND**, then **NOT**, then the relations and the usual operations.

NOT is really a function, with an argument and a res. ll, but its priority is much lower than that of other functions. Therefore its argument does not need parentheses unless it contains **AND** or **OR** or both: **NOT a=b** means the same as **NOT (a=b)** and the same as **a<>b** (of course).

<> is the negation of **=** in the sense that it is true if, and only if, **=** is false. (In other words,

a<>b is the same as **NOT a=b**

and also

NOT a<>b is the same as **a=b**

Persuade yourself that **>=** and **<=** are the negations of **<** and **>** respectively, thus you can always get rid of **NOT** from in front of a relation by changing the relation.

Also

NOT (a first logical expression AND a second)

is the same as

NOT (the first) OR NOT (the second)

and

NOT (a first logical expression OR a second)

is the same as

NOT (the first) AND NOT (the second)

Using this, you can work NOTs through parentheses until eventually they are all applied to relations, and then you can get rid of them. Logically speaking **NOT** is unnecessary, although you might still find that using it makes a program clearer.

The following section is quite complicated, and can be skipped by the fainthearted!

Try

```
PRINT 1=2, 1<>2
```

which you might expect to give a syntax error. In fact, as far as the computer is concerned there is no such thing as a logical value. Instead it uses ordinary numbers, subject to a few rules:

1. =, <, >, <=, >= and <> all give numeric results: 1 for *true* and 0 for *false*. Thus the **PRINT** command above printed 0 for 1=2, which is *false*, and 1 for 1<>2, which is *true*.
2. **IF condition THEN** — the condition can be actually any numeric expression. Its value is 0 then it counts as *false*, and any other value (including the value of 1, that a *true* relation gives) counts as *true*. Thus the **IF** statement means exactly the same as **IF condition <> 0 THEN**.
3. **AND**, **OR** and **NOT** are also number valued operations.

x AND y has the value

{	x if y is true (non-zero)
{	0 (false) if y is false (zero)

x OR y has the value

{	1 if x or y is true (non-zero)
{	x if y is false (zero)

NOT x has the value

{	0 (false), if x is true (non-zero)
{	1 (true), if x is false (zero)

Notice that *true* means non-zero when we're checking a given value, but it means 1 when we're producing a new one.

Read through the chapter again in the light of this revelation, making sure that it all works.

In the expressions **x AND y**, **x OR y** and **NOT x**, x and y will usually take the values 0 and 1 for *false* and *true*. Work out the ten different combinations, four for **AND**, four for **OR** and two for **NOT**, and check that they do what the chapter leads you to expect them to do.

Try this program

```
10 INPUT a
20 INPUT b
30 PRINT (a AND a>=b) + (b AND
  a<b)
40 GO TO 10
```

Each time it prints the larger of the two numbers a and b. Convince yourself that you can think of

x AND y as meaning: x if y (else the result is 0)
and of
x OR y as meaning: x unless y (in which case the result is 1)

An expression using **AND** or **OR** like this is called a *conditional expression*.

An example using **OR** could be

```
price=price_ess_tax* 1.15 OR v$='zero
rated )
```

Notice how **AND** tends to go with addition (because its default value is 0) and **OR** tends to go with multiplication (because its default value is 1).

You can also make string valued conditional expressions but only using **AND**.

$x\$ \text{ AND } y$ has the value $\begin{cases} x\$ & \text{if } x\$ \neq "" \text{ and } y \neq "" \\ "" & \text{if } x\$ = "" \text{ or } y = "" \end{cases}$

So it means $x\$$ if y else the empty string.

Try this program which inputs two strings and puts them in alphabetical order.

```
10 INPUT 'Type in two
   strings a$,b$
20 IF a$>b$ THEN
   c$=a$ a$=b$ b$=c$
30 PRINT a$ " ", (" "<" AND a$
   <b$)+ " = " AND a$=b$ ,
   ' ', b$
40 GO TO 10
```

Exercise

NextBASIC can sometimes work along different lines from English. Consider for instance the English clause "a doesn't equal b or c". How would you write this in *NextBASIC*? The answer is not

```
IF a<>b OR c
not is
IF a<>b OR a<>c
```


Chapter 13 The Character Set

The letters, digits, punctuation marks and so on that can appear in strings are called *characters*, and they make up the alphabet, or character set that the ZX Spectrum Next uses.

CHR\$ and CODE

As you will also see in *Appendix A*, there are 256 character locations, and each one is assigned a code between 0 and 255. To convert between codes and characters, two functions exist: **CODE** and **CHR\$**. **CODE** is applied to a string, and returns the code of the first character in the string (or 0 if the string is empty). **CHR\$** is applied to a code, and returns the single character string that corresponds to that code. We started this paragraph by saying ‘character locations’ and not simply ‘characters’. As we will find out further below, and in *Chapters 14–20* and *Appendix A*, some characters are non-printable and as a matter of fact perform special functions. The following little program prints out the entire usable character set.

```
10 FOR a=32 TO 255 PRINT CHR$ a NEXT a
```

At the top you can see a space, 5 symbols and punctuation marks, the ten digits, seven more symbols, the capital letters, six more symbols, the lower case letters and five more symbols. These are all (except **£** and **@**) taken from a widely-used set of characters known as *ASCII* (standing for *American Standard Codes for Information Interchange*). *ASCII* also assigns numeric codes to these characters, and these are the codes that the ZX Spectrum Next uses.

The graphics symbols

The rest of the characters are not part of *ASCII*, and are specific to the ZX Spectrum Next. First amongst them are a space and 5 patterns of black and white pixels. These are called the *graphics symbols* and can be used for drawing rudimentary pictures. You can enter these from the keyboard, using what is called *graphics mode*.

If you press **GRAPHICS** then the cursor will change to a flashing white/magenta. Now the keys for the digits 1 to 8 will give the graphics symbols. In their own they give the symbols drawn on the keys, and with either shift pressed, they give the same symbol but inverted (i.e. black becomes white, and vice versa). Regardless of shifts, digit 9 takes you back to normal mode (blue cursor) and digit 0 is **DELETE**. Here are the sixteen graphics symbols.

















Symbol	Code	Key	Symbol	Code	Key
	128	8		148	Shift+8
	129	9		149	Shift+9
	130	2		145	Shift+2
	131	3		140	Shift+3
	132	4		139	Shift+4
	133	5		138	Shift+5
	134	6		137	Shift+6
	135			136	Shift+7

Table 5 Graphics Symbols

Tokens

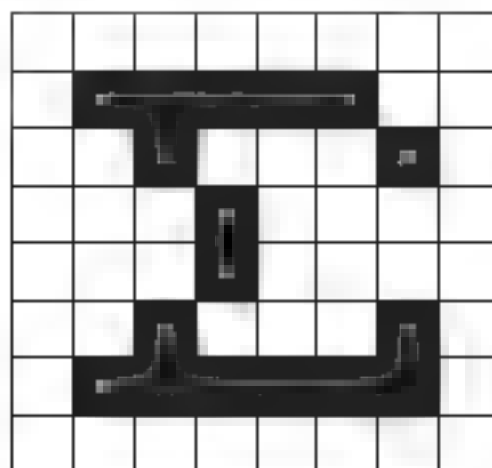
When not used as single symbols, several character codes are used in an alternative manner where they are called *tokens*. Tokens represent whole words, such as **PRINT**, **STOP**, **>=**, **<>**, **<=** and so on. This is to save space in the machine's main RAM, thus maximising the available program space by substituting multiple character words for single character codes.

BIN and JSR

After the graphics symbols, you will see what appears to be another copy of the alphabet from **A** to **z**. These are characters that you can redefine yourself, although when the machine is first switched on they are set as letters. They are called *user-defined graphics* or **UDGs** for short. You can type these in from the keyboard by going into graphics mode and then using the letters keys from **A** to **z**.

To define a new character for yourself, follow this recipe: it defines a character to show the mathematical symbol Σ (Greek for $\Sigma\upsilon\nu\alpha\lambda\omicron$ =sum).

Work out what the character looks like. Each character has an 8x8 square of dots, each of which can show either the paper colour or the ink colour (see Chapter 15 regarding **INK** and **PAPER**). You'd draw a diagram something like this, with black squares for the ink colour.



We've left a 1 square margin round the edge because the other letters all have one (except for lower case letters with 'ails' where the tail goes right down to the bottom of the square).

- i Work out which user-defined graphic is to show (let's say the one corresponding to **S**, so that if you press **S** in graphics mode you get Σ on your screen).
- ii Store the new pattern. Each user-defined graphic has its pattern stored as eight numbers, one for each row. You can write each of these numbers as **BIN** followed by eight 0s or 1s: 0 for paper, 1 for ink, so that the eight numbers for our character are

```

BIN 00000000
BIN 01111100
BIN 00100010
BIN 00010000

```



```

BIN 00010000
BIN 00100010
BIN 01111110
BIN 00000000

```

If you know about binary numbers, then it should help you to know that **BIN** is used to write a number in binary instead of the usual decimal.

These eight numbers are stored in memory in eight places, each of which has an address. The address of the first byte, or group of eight digits, is `USR "S" 'S` because that is what we chose in (ii). That of the second is `USR "S" + 1` and so on up to the eighth, which has address `USR "S" + 7`.

USR here is a function to convert a string argument into the address of the first byte in memory of the corresponding user-defined graphic. The string argument must be a single character which can be either the user-defined graphic itself or the corresponding letter in upper or lower case. There is another use for **USR** when its argument is a number which will be dealt with in subsequent chapters.

Even if you don't understand this, the following program will do it for you.

```

5 FOR n=0 TO 7
10 READ row, POKE USR
   S"+n, row
15 NEXT n
20 DATA BIN 00000000
25 DATA BIN 01111110
30 DATA BIN 00100010
35 DATA BIN 00010000
40 DATA BIN 00010000
45 DATA BIN 00100010
50 DATA BIN 01111110
60 DATA BIN 00000000

```

The above example can also be rewritten using integer variables without the use of **BIN** while still expressing the graphic matrix in binary form. Can you restate it per what you've learned?

POKE and PEEK

The **POKE** statement stores a number directly in a memory location, bypassing the assignment (**LET**) mechanism normally used by **NexiBASIC** which also tracks its place in memory. The opposite of **POKE** is **PEEK** and this allows us to look at the contents of a memory location although it does not actually alter the contents of that location. They will be dealt with properly in Chapter 23. There are a few more efficient ways to type all the above but for now, we're using the simplest forms of **PEEK** and **POKE**.

The tokens (which we referred to a little earlier) are stored right after character code 128. As you saw in the character set printing example, codes 0 to 31 were absent. These are control characters or as commonly referred to, control codes. They either don't produce characters on screen (although they do have an effect on what's printed there) or alternatively they are used to control something other than the display itself and so screen displays " ? " to show that it doesn't understand them. They are described more fully in Appendix A.

Three control codes that are used with screen output are those with codes 6, 8 and 13.

CHR\$ 6 prints spaces in exactly the same way as a comma does in a PRINT statement or instance

```
PRINT 1, CHR$ 6 2
```

does the same as

```
PRINT 1,2
```

Obviously this is not a very clear way of using it. A more subtle way is to say

```
a$='1'+CHR$ 6+ '2'
PRINT a$
```

CHR\$ 8 is *backspace*. It moves the print position back one place. try

```
PRINT "1234", CHR$ 8, '5'
```

which prints up

```
1235
```

as 5 takes the place of 4 from the string printed in the first part of the PRINT statement.

CHR\$ 13 is *carriage return*. It moves the print position on to the beginning of the next line.

Effectively

```
PRINT "1234", CHR$ 13, "5678"
```

is the same as

```
PRINT "1234"
PRINT "5678"
```

may not be immediately apparent why you wouldn't do the latter but it's possible also to do

```
a$="1234"+CHR$ 13+ "5678"
PRINT a$
```

in which case you can see the usefulness of a single *carriage return* character.

The screen also uses control codes 16 to 23. These are explained in *Chapters 13 and 14*. All the control codes are listed in *Appendix A*.

Using the codes for the characters we can extend the concept of *alphabetical ordering* to cover strings containing any characters, not just letters. If instead of thinking in terms of the usual alphabet of 26 letters we use the extended alphabet of 256 characters in the same order as their codes, then the principle is exactly the same. For instance, these strings are in their ZX Spectrum Next alphabetical order. (Notice the rather odd feature that lowercase letters come after all the capitals, so a comes after Z, also, spaces matter)

```
CHR$ 3+"ZOOLOGICAL GARDENS"
CHR$ 8+"AARDVARK HUNTING"
"AAAARGH!"
"(Parenthetical remark)"
"100"
"$129.95 inc VAT"
"AASVOGEL"
"Aardvark"
"PRINT"
"Zoo"
```



```
"[interpolation]"
"aardvark"
"aasvogel"
"zoo"
"zoology"
```

Here is the rule for finding out which order two strings come in. First, compare the first characters. If they are different, then one of them has its code less than the other, and the string that came from it is the earlier (lesser) of the two strings. If they are the same, then go on to compare the next characters. If, in this process, one of the strings "runs out" before the other, then that string is the earlier; otherwise they must be equal.

The relations `=`, `<`, `>`, `<=`, `>=` and `<>` are used for strings as well as for numbers. `<` means comes before and `>` means comes after, so that:

```
"AA man"<"AARDVARK"
"AARDVARK">"AA man"
```

are both true.

`<=` and `>=` work the same way as they do for numbers, so that

```
"The same string"<="The same string"
```

is true, but

```
"The same string"<"The same string"
```

is false.

Experiment on all this using the program here, which inputs two strings and puts them in order.

```
10 INPUT "Type in two
   strings.", a$, b$
20 IF a$>b$ THEN a$=b$,a$
30 PRINT a$ " ",
40 IF a$<b$ THEN PRINT "<",
   GO TO 60
50 PRINT " = "
60 PRINT " ", b$
70 GO TO 10
```

Note how we are using a multiple assignment in order to swap `a$` and `b$` in line 20 as

```
a$=b$ b$=a$
```

would not have the desired effect, since `a$` would have the value of `b$` prior to trying to assign its value to `b$`.

This program sets up user-defined graphics to show chess pieces.

```
P for pawn
R for rook
N for knight
B for bishop
K for king
Q for queen
```


Chess pieces

```

5  b,c,d=BIN 01111100,BIN
   00111000,BIN 00010000
10  FOR n=1 TO 5  READ p$  REM
   5 pieces
20  FOR f=0 TO 7: REM read
   piece into 8 bytes
30  READ a  POKE LSR p$+f,a
40  NEXT f
50  NEXT n
100 REM bishop
110 DATA "b",0,d, BIN
   00101000,BIN 01000100
120 DATA BIN 01101100 c,b,0
130 REM king
140 DATA "k",0,d,c,d
150 DATA c, BIN 01000100,c,0
160 REM rook
170 DATA "r",0, BIN
   01010100,b,c
180 DATA c b,b,0
190 REM queen
200 DATA "q",0, BIN 01010100,
   BIN 00101000,d
210 DATA BIN 01101100,b,b,0
220 REM pawn
230 DATA "p",0,0,d,c
240 DATA c,d,b,0
250 REM knight
260 DATA "n",0,d,c, BIN
   01111000
270 DATA BIN 00011000 c,b,0

```

Note that 0 can be used instead of BIN 00000000

When you have it in the program look at the pieces by going into graphics mode

Alternative Character Sets

As we are going to see in Chapter 20 Channels, Streams and Windows the ZX Spectrum Next provides via its windowing system the ability to display alternative character sets. In order to set up however an alternative character set characters have to be defined somewhere in memory very similarly to the way we did the chess pieces or the Z symbols above. The characters redefined are limited in the 96 from code 32 until code 127 and should be in that order. A successive series of 768 POKE statements incrementing the memory address by one location at the time will define them and then a fast POKE altering the

CHARS system variable. See *Chapter 24 System variables* will point NextBASIC to the location of this new character set.

Character Graphics Mode

In the following chapter we will be introduced to Layer 3 – the Character Graphics mode. This is a hybrid graphics mode based around the notion of a character tile (that is to say an 8 × 8 pixel matrix very much like the ones we explored above with User Defined Graphics) with four very crucial differences:

- Each character tile can have up to sixteen colours and not only two
- All ASCII characters can be defined by tiles giving the user in effect a truly multi-lingual character display
- Layer 3 displays can be either 80 columns by 32 rows or 40 columns by 32 rows and not only 32 columns by 24 rows as the regular Spectrum display is
- Layer 3 cannot be accessed from NextBASIC (at the time of writing) in the same straightforward way other modes/layers are. You will need to write functions and procedures that utilise the PEEK, POKE, IN, OUT and REG facilities as well as the BANK commands at your disposal in order to make use of this powerful mode.

Layer 3 has other uses as well and we will be discussing those in the following 3 chapters.

Exercises

1. Imagine the space for one symbol divided up into four quarters like a Battenberg cake. Then if each quarter can be either black or white there are $2 \times 2 \times 2 \times 2 = 16$ possibilities. Find them all in the character set.

2. Run this program

```
10 INPUT a
20 PRINT CHR$ a
30 GO TO 10
```

If you experiment with it you'll find that CHR\$ a is rounded to the nearest whole number and if a is not in the range 0 to 255 then the program stops with error report

B integer out of range

3. Which of these two is the lesser?

```
"EVL"
"evil"
```


Chapter 14 More about PRINT and INPUT

Coordinate Systems

[illegible]

Screen Modes and Pixel Coordinates

Now, to make a correct distinction between the two card 3D systems, we should distinguish between the two cases: $2X = 2Y$ and $2X \neq 2Y$. We will consider the drawing of figure 15 as an example of a 3D representation of a 3D object. In this case, the two systems are identical, and the two systems are identical. But for now let's enumerate the screen modes in a simple fashion:

The ZX Spectrum has a data logging facility to store data in groups or arrays. We will illustrate this with an example which will store 10000 values. These values are processed using the `LAYER` command with the expression `0.0001 * value`. The `GROUP` is called `TEMP` and the data is stored in a file called `TEMP.DAT` as follows:

- Layer 0
 - ▶ Layer 0.0 Standard Ras (Enhanced JLA, mode: 256 w x 192 h pixels, 256 colours total) 32 x 24 cells, each capable of displaying 2 colours
- Layer 1
 - ▶ Layer 1.0 LoRes (EnhancedULA, mode: 128 w x 96 h pixels, 256 colours total) 1 colour per pixel
 - ▶ Layer 1.1 Standard Ras (Enhanced JLA, mode: 256 w x 192 h pixels, 256 colours total) 32 x 24 cells, each capable of displaying 2 colours
 - ▶ Layer 1.2 Timex HiRes (EnhancedULA, mode: 512 w x 192 h pixels, 256 colours total, only 2 colours on screen)
 - ▶ Layer 1.3 Timex HiColour (EnhancedULA, mode: 256 w x 192 h pixels, 256 colours total) 32 x 192 cells, each capable of displaying 2 colours
- Layer 2
 - ▶ Layer 2.0 256 w x 192 h pixels, 256 colours total, one colour per pixel
 - ▶ Layer 2.2 320 w x 256 h pixels, 256 colours total, one colour per pixel
 - ▶ Layer 2.3 640 w x 256 h pixels, 16 colours total, one colour per pixel
- Layer 3
 - ▶ Layer 3.0 Text mode: 320 w x 256 h pixels, 256 colours total, 40 x 32 cells, each capable of displaying 2 colours
 - ▶ Layer 3.1 Text mode: 640 w x 256 h pixels, 256 colours total, 80 x 32 cells, each capable of displaying 2 colours
 - ▶ Layer 3.2 Graphics mode: 320 w x 256 h pixels, 256 colours total, 40 x 32 cells, each capable of displaying 16 colours
 - ▶ Layer 3.3 Graphics mode: 640 w x 256 h pixels, 256 colours total, 80 x 32 cells, each capable of displaying 16 colours

INPUT and output are not discussed in this paper. The main reason for this is incompleteness.

Technically speaking, Layer 1 is the same as Layer 0 with extra colour capabilities. However NextBASIC treats them differently to maintain a consistent way of addressing the extra capabilities of the ZX Spectrum Next's *EnhancedULA*. The legacy coordinate system we discussed above applies only to Layer 0, whereas Layers 1 and 2 use the new system.

There are three major differences between Layer 0 and Layers 1 and 2 as far as character positioning goes. There are more differences but we will examine these in turn in the special graphics Chapters 15–17. These are:

1. Layer 0 is organised in a strict 32 columns by 24 rows matrix while the rest can both position characters on a similar matrix according to character size, or, if so desired, anywhere along the y and x axes.
2. The user cannot normally position characters on the two bottom rows of the Layer 0 screen while this is possible in the other layers.
3. Layer 0 pixel coordinates begin at the bottom left corner and extend up and to the right while for the rest of the layers pixel coordinates begin at the top left corner and extend down and to the right. This particular difference is not important for character placement on Layer 0 but it is for the rest of the layers and is timely as we are going to see further down this manual extremely important for positioning graphics.

Changing the size of characters

With the exception of Layer 0, which has, as we mentioned, a rigid organisation of character positions on screen in a 32 x 24 character matrix, all other layers have the ability to position characters either rigidly as above, ie in a rows x columns matrix, or freely according to pixel position of each character matrix's top left corner.

Character size can be modified horizontally with the following sequence:

```
PRINT CHR$ 30: CHR$ n
```

where *n* can be a number from 3 to 8 which sets the width of all characters displayed on screen from a minimum of 3 to a maximum of 8 pixels wide. Character size is modified vertically by issuing:

```
PRINT CHR$ 29: CHR$ n
```

where *n* can be a number from 0 to 3 which sets the height of all characters displayed on screen to the following predetermined heights in pixels:

Value of <i>n</i>	Size (pixels)	Description
0	8	Normal Size
1	16	Double Size
2	6	Reduced Size
3	12	Double Reduced Size

These sequences which are more appropriately called control codes, are character size shortcuts for text windows. These can also be used on Layer 0 but you would need to open a window first when in that mode. The rest of the layers have predefined and pre-coded full-screen text windows and therefore these control codes work here by default. We will discuss text windows at length in Chapter 20: Channels, Streams and Windows so for now keep these two control codes in mind as only working outside Layer 0. They are extremely important to know as they modify the behaviour of the AT and TAB modifiers we will examine below.

Using AT to print to a certain location

You have already seen PRINT used quite a lot, so you will have a rough idea of how it is used. Expressions whose values are printed are called PRINT items and they are separated by commas, semicolons and apostrophes which are called PRINT separators. A PRINT item can also be nothing at all which is a way of explaining what happens when you

Table 6 above showed us that although we could pack our screen with 170 characters per line, in practice 3 pixel-wide fonts are almost unreadable even at the highest available resolution of Layer 2. In the example program that's meant to demonstrate character cells for the AT modifier (but written using the POINT modifier), amazingly enough, we're including below you can see all the possible combinations for all layers.

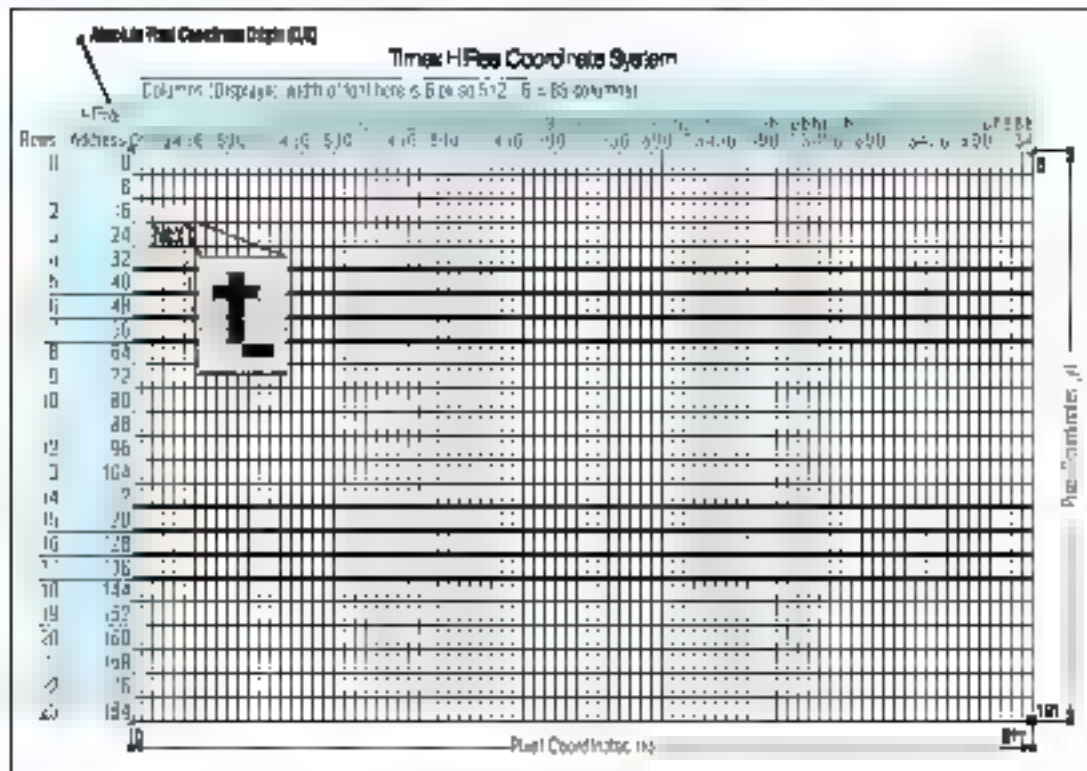


Fig. 8 Layer 0 coordinate system for PRINT and INPUT

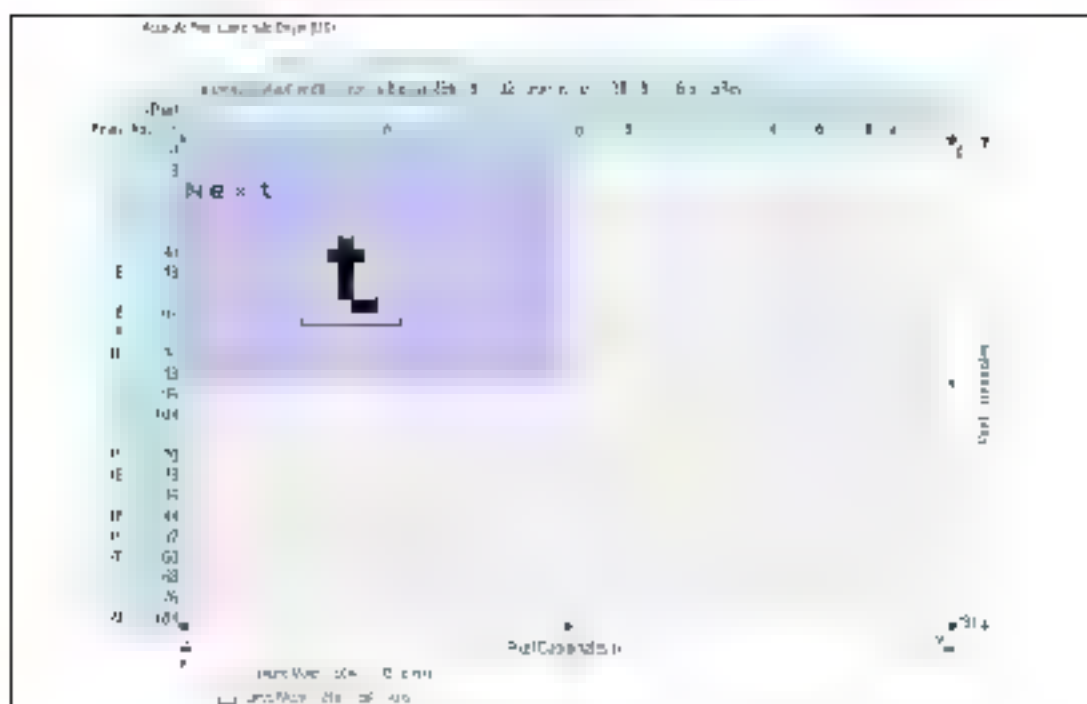


Fig. 9 HiRes and Standard Resolution coordinate system for PRINT and INPUT

The author's personal preference is the 128 column text of HiRes Layer 2 as it's clear enough to read but not too big as to not be able to fit a lot of information onto your screen.

Using POINT to print to a certain location

In Fig. 9 above, we see the main difference between *PRINT* items on Layer 0 and the other layers and that's none other than the previously mentioned ability to place them in any X and Y coordinate we please. This diagram assumes a standard 8x8 character size but where you only saw rows in Fig. 8, here you also see a pixel value. This corresponds to the placement of each row and column in Layer 0 print in fact, it could be anything within the boundaries of the horizontal and vertical resolution. Let's switch layers and try to do the same thing.

```
LAYER 1,1 PRINT POINT 248,176, "*"
```

Unlike before you'll will not get an **5 Out of screen, 0 1** error and you will get an asterisk at the rightmost edge of the screen like we expected to get the first time we gave the **PRINT AT 22,31** command. The two values correspond to 22 times the character height and 31 times the character width (both of which are 8 pixels). You can see at the same time the notion of the free placement of characters as the addressing of the location is now in pixels and not the fixed rows and columns. What's also immediately visible is that addressing the location of screen in pixel coordinates is different as it reverses the order of the location parameters from *y x* to *x y* and that's done to match the syntax of the rest of the graphics commands that accept pixel coordinates as parameters. To replicate the behaviour of the first **PRINT AT** command on Layer 0 and get an error, we will need to place the output of print outside the boundaries of the screen like so

```
LAYER 1,1 PRINT POINT 256,0, "*"
```

would produce the same exact error. To properly calculate where to print if you want to keep your coordinates cell-based instead of pixel-based, a simple function could do that for you quite easily. In Fig. 8 as well as Fig. 9 we've done that for you assuming a standard font but what about a shorter or perhaps taller font? It's quite simple if you keep in mind that if you follow the heights defined earlier, you can find exactly how many rows and columns you can fit in your screen. Note that **POINT**'s arguments must not begin with a parenthesis because it will be evaluated as a function and attempting to store the line you're typing will fail.

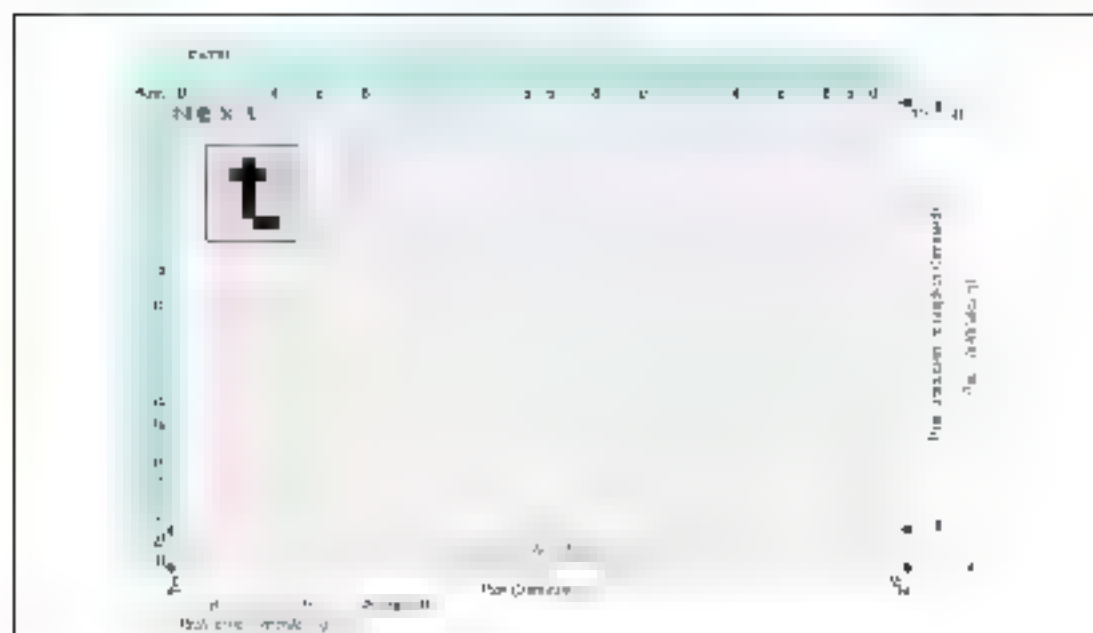


Fig. 10 High Resolution coordinate system for PRINT and INPUT

The following very slow- program demonstrates exactly how things are positioned on screen with every change in layer and furthermore gives you some insight on how **PRINT POINT** as well as indirectly- **PRINT AT** is affected every time your screen mode changes. Try to walk through the program to figure out how it operates.


```

10 REM First we disable LAYER
   2 and then we set Standard
   JLA Display Mode
20 LAYER 2 0
30 LAYER 0
40 MaxX,MaxY=128,96
70 mul,div,add=1,1,0
100 chsz,h=8
120 FOR m=0 TO 5
130 n,d=0,1
150 IF m=0 THEN GO TO 370 REM
   Layer 0 not supported by
   PRINT POINT
160 FOR a=3 TO 8
180 FOR b=0 TO 3
190 n,d=0,1
210 PROC LayChange(m a,b)
220 FOR r=0 TO (MaxY*mul) 1
   STEP h
230 FOR c=0 TO (MaxX*m) chsz
   STEP chsz
235 row=n+(r+add)/div
240 IF r=0 AND c<>0 THEN PRINT
   POINT
   c,row,d d+=1
250 IF c=0 AND r=0 THEN PRINT
   POINT
   c,row,n n+=1
260 IF c=0 AND r<>0 THEN PRINT
   POINT
   c,row,n n+=1
270 IF c<>0 AND r<>0 THEN
   PRINT POINT
   c,row,'*'
280 IF n=10 THEN n=0
290 IF d=10 THEN d=0
300 NEXT c
310 IF c=1 THEN n=0
320 NEXT r
330 PALSE 0
340 IF m=0 THEN GO TO 370
350 NEXT b
360 NEXT a

```



```

370 NEXT m
380 LAYER 0
390 LAYER 2 0
400 STOP

1000 DEFPROC LayChange mode,ch,he1
1010 div,add,maxX,maxY,mul,
    chsz=1 0,128,96 2 ch
1070 IF he=0 THEN h=8
1080 IF he=1 THEN h=16
1090 IF he=2 THEN h=8
1100 IF he=3 THEN h=12
1110 REM Layer 0 is not covered
    as PRINT POINT doesn't work
1120 IF mode=1 THEN LAYER 1,0
    CLS mul=1 PRINT CHR$ 30,
    CHR$ ch PRINT CHR$ 29, CHR$
    he PRINT AT 0,0,'LoRes"'
    CSIZE (HXW) "h," x
    ",chsz'"PRESS ANY KEY"
    PAUSE 0 CLS ENDPROC
1130 IF mode=2 THEN LAYER 1,1
    CLS PRINT CHR$ 30, CHR$
    ch PRINT CHR$ 29,CHR$ he
    PRINT AT
    0,0,'EnhancedULA' 'CSIZE
    (HXL) "h,
    " x ",chsz'"PRESS ANY KEY"
    PAUSE 0 CLS ENDPROC
1140 IF mode=3 THEN LAYER 1,2
    CLS maxX=255 PRINT CHR$ 30,
    CHR$ ch PRINT CHR$ 29, CHR$
    he PRINT AT 0,0,'Timex
    HiRes 'CSIZE (HXW ,h,
    x "chsz 'PRESS ANY KEY'
    PAUSE 0 CLS ENDPROC
1150 IF mode=4 THEN LAYER 1,3
    CLS PRINT CHR$ 30, CHR$ ch
    PRINT CHR$ 29, CHR$ he
    PRINT AT 0,0;"Timex
    HiColour" CSIZE (HXW)
    ",h," x ",chsz'"PRESS ANY
    KEY" PAUSE 0 CLS ENDPROC

```



```

1160 IF mode=5 THEN LAYER 2,1
      CLS PRINT CHR$(70) CHR$(
      ch PRINT CHR$(29), CHR$(
      he PRINT AT
      0,0, Layer(2)""C$IZE (HxW)
      ",h," x ",chsZ""PRESS ANY
      KEY PAUSE 0 CLS ENDPROC

```

SCREEN\$

SCREEN\$ is the reverse function to **PRINT AT** and will, if you wish, return what character is at a particular position on the screen. It uses line and column numbers in the same way as the *Layer 0* version of **PRINT AT** but enclosed in parentheses. For instance

```
PRINT SCREEN$ (11,16)
```

will retrieve the star you printed in the first example of the previous section. **SCREEN\$** only works on *layer 0* and will return everything it finds there even if you switch layers during the process as long as the memory used (which is shared between *Layers 0-7* and *1* as you will see in Chapter 24) has not been overwritten by another display-related command type.

```

10 LAYER 0 PRINT AT 11,11, *
20 LAYER 1 0 PRINT AT
   0,0,SCREEN$ (11,11)

```

You will get a huge * on the upper left corner of your screen even if the original * is not visible anymore on screen. Changing line 10 to **LAYER 1 0** from **LAYER 0** will produce a null string.

Characters taken from tokens print normally, as single characters, and spaces return as spaces. Lines drawn by **PLOT DRAW** or **CIRCLE**, user-defined characters and graphics characters return as a null (empty) string, however, the same applies. **OVER** (See Chapter 6) has been used to create a composite character. The way that **SCREEN\$** works is that it matches the character in a screen location to the bit mapped image of the character in the ROM of NextZX05. If they match it will return it, if the picture in the location doesn't match any known character it will return an empty string.

TAB

If you're familiar with word processing, other computers, or even typewriters, you may be also familiar with the concept of a *tab* or *tabulating* character. What this does in other computers is to insert a special character which will move the cursor right by a predetermined amount, or locations in other words, to a specific column in your text. The ZX Spectrum Next doesn't quite work like this although the ending result on your screen is pretty much equivalent. The modifier

TAB column

prints enough spaces to move the **PRINT** position to the column specified. It stays on the same line, so if this would involve backspacing, moves on to the next one. Note that the computer reduces the column number modulo *X* with *X* being the maximum amount of columns available per the width of character (32 spaces for each layer, meaning 32 divides by *X* and takes the remainder). So for example for *Layer 0* **TAB 33** means the same as **TAB 1**.

The code

```
PRINT TAB $(1,TAB 12,"Contents ", AT
3 1, 'CHAPTER ',TAB 24, page')
```


demonstrates how you might print out the heading of a contents page on page 1 of a book if that book was displayed using ZX Spectrum Next characters of course!

Try running this

```
10 FOR n=0 TO 20
20 PRINT TAB 8+n,n,
30 NEXT n
```

This shows what is meant by the **TAB** numbers being reduced modulo *X*. For a more elegant example, change the 8 in line 20 to a 6 or even try to implement this on a different layer such as the *Holes* one as it allows more room for demonstration of this functionality by adding **LAYER 1.2** before line 10.

As you¹ see in Chapter 20, **TAB** accepts a two-byte parameter which means it accepts a maximum column number of 65536. Not that you'd ever want to use that.

Some small points

- 1 These new items are best terminated with semicolons, as we have done above. You *can* use commas, or nothing, at the end of the statement, but this means that after having carefully set up the **PRINT** position, you immediately move it on again which wouldn't usually be terribly useful.
- 2 As a reminder, you cannot print on the bottom two rows (22 and 23) on the *Layer 0* screen because they are reserved for commands, **INPUT** data, set below, reports/errors and so on. References to the *bottom* line usually mean line 21 and only apply to *Layer 0*.
- 3 You can use **AT** to put the **PRINT** position even where there is already something printed: the old stuff will be obliterated when you print more.

CLS

Another statement that's connected with **PRINT**, although it's not only limited to it, is **CLS**.

This clears ~~the whole screen~~, something that is also done by **CLEAR** and **RUN**. The **LAYER** command does not clear the screen however, although it may switch to a new screen that has nothing on it. Do not assume a *Layer* is free of stuff just because you haven't used a command that outputs something ~~on screen~~. Always give **CLS** after switching layers if you want to ensure a screen free of anything on it.

Scrolling

When the printing reaches the bottom of the screen, the latter moves its contents upwards to clear room on the bottom for new content. You can see this if you go into the status area, by using the *Edit menu* option *Screen* and then type

```
CLS FOR n=1 TO 22 PRINT n NEXT n
```

and then do

```
PRINT 99
```

a few times.

Depending on the *layer* you are on, the computer may pause its screen output for you to view the content being printed and ask you a question or may simply display a block cursor at the lower right corner and wait.

On *layer 0*, if the computer is printing big teams and teams of stuff on screen, it asks you, before continuing. You can see this happening if you type

```
CLS FOR n=1 TO 100 PRINT n NEXT n
```


When it has printed a screenful, it will stop, writing *scroll?* at the bottom of the screen. You can now inspect the first 22 numbers at your leisure. When you have finished with them, press *y* (for 'yes') and the computer will give you another screenful of numbers. Actually any key will make the computer carry on, except *n* (for 'no'), **SYMBOL SHIFT** and **A** (for **STOP** as you can see printed on your ZX Spectrum Next's keyboard²). **SPACE BREAK** (or **CAPS SHIFT** and **SPACE** or **Esc**, the latter if you have a PS/2 type keyboard). These will make the computer stop running the program with a report **DBREAK CONT repeats**. On other layers, the *scroll?* message is replaced by a block cursor, called the *scroll prompt cursor*, at the lower right corner. The only keys which will stop the scrolling in layers other than 0 are the **Esc** key (on a PS/2 keyboard) or the **BREAK** key (**CAPS SHIFT** and **SPACE**). Everything else will scroll the screen.

Expanding on INPUT

The **INPUT** statement can do much more than we have told you so far. You have already seen **INPUT** statements like

```
INPUT "How old are you?", age
```

in which the computer prints the caption *How old are you?* at the bottom of the screen and then you have to type in your age.

In fact, an **INPUT** statement is made up of items and separators in exactly the same way as a **PRINT** statement is, so *How old are you?* and *age* are both *INPUT* items. *INPUT* items are generally the same as *PRINT* items, but there are some very important differences.

First, an obvious extra *INPUT* item is the variable whose value you are to type in, *age* in our example above. The rule is that if an *INPUT* item begins with a letter, it must be a variable whose value is to be input.

Second, this would seem to mean that you can't print out the values of variables as part of a caption, however you can get round this by putting parentheses around the variable. Any expression that starts with a letter must be enclosed in parentheses if it is to be printed as part of a caption.

Any kind of *PRINT* item that is not affected by these rules is also an *INPUT* item. Here is an example to illustrate what's going on.

```
myage=INT RND 100,) INPUT ("I am ",myage,
,"); How old are you? , yourage
```

myage is contained in parentheses, so its value gets printed out, *yourage* is not contained in parentheses, so you have to type its value in.

If you are in Layer 0, everything that an **INPUT** statement writes goes to the bottom part of the screen, which acts somewhat independently of the top half. In particular, its rows are numbered relative to the top line of the bottom half, even if this has scrolled the actual screen up (which it does if you type lots and lots of **INPUT** data).

To see how **AT** works in **INPUT** statements, try running this on Layer 0:

```
10 INPUT 'This is line
1.",a$, AT 0,0,'This is
line 0.",a$, AT 2 0
This is line 2," a$, AT
1,0, "This is still line
1.",a$
```

² This functionally comes from the original ZX Spectrum single key (in intended) entry and it's retained for compatibility reasons.

just press **ENTER** each time it's done. When this is line 2 is printed, the lower part of the screen moves up to make room for it, but the numbering moves up as well, so that the rows of text keep their same numbers.

Now try this (again on layer 0):

```
10 FOR n=0 TO 10 PRINT AT
   n,0,n, NEXT n
20 INPUT AT 0,0,a$, AT 1,0,a$,
   AT 2,0 a$, AT 3,0 a$, AT
   4,0,a$ AT 5,0,a$
```

As the lower part of the screen scrolls up and up, the upper part is undisturbed and the lower part in fact ends up with one line more as the **PRINT** position. Then the upper part starts scrolling up to avoid this.

The other layers work in the same manner as described for **PRINT** items: that is in both single cell matrix and flexible like coordinate terms. Illustrate the difference: issue a **LAYER 11** clear command and then modify the first example by first copying line 10 to line 20 and then changing all **AT** statements to **POINT** statements switching the x and y positions around, thus making the latter use parameters 0-16 and 0-8 respectively. The local height of characters is more than on layers other than 0: character matrices will change according to character size and fixed positioning according to max resolution. The first thing you'll notice is that **INPUT** takes place at the top left of the screen as would with **PRINT** and the second one that the first **INPUT** item is **N0** printed at line number 0. In line 0, finally, you can see from the modified first example that **INPUT** accepts a **POINT** modifier for positioning exactly like **PRINT** does.

LINE input

Another refinement to the **INPUT** statement that we haven't seen yet is called **LINE** input and is a different way of inputting string variables. You write **LINE** before the name of a string variable to be input, as in

```
INPUT LINE a$
```

then the computer will not give you the string quotes that it normally does for a string variable although it will pretend to itself that they are there. So if you type in

```
Simon
```

as the **INPUT** data **a\$** will be given the value **Simon**. Because the string quotes do not appear on the string, you cannot directly read and type in a different sort of string expression in the **INPUT** data. Remember that you cannot use **LINE** for numeric variables.

Using Expressions for INPUT

There's an interesting capability of **INPUT**. While typing into an **INPUT** requires that's expecting a numeric variable, you can use numeric expressions which can include previously defined variables. Try running this program:

```
10 a=14
20 INPUT numbers
30 PRINT numbers
40 GO TO 20
```

Input a few numbers, and they'll be printed as expected on the screen. Now type **a** and if you press **ENTER** then **14** will appear. Try typing **a+2** and **16** will appear. However, if you

type a variable name not previously defined – and the computer will stop with the report **2 Variable not found. 20**

Using control codes with PRINT

In the beginning of this chapter we saw the effect that control codes 28 and 30 had in adjusting the size of the font that's currently printed on screen. There are more control codes that we can use with PRINT. CHR\$ 22 and CHR\$ 23 affect printing in the same manner as AT and TAB. They are rather odd as control codes, because whenever one is sent to the screen to be printed, it must be followed by two more characters that do not have their usual effect: they are treated as numbers. The codes 22 specify the y and x positions (the AT) or the tab position (or TAB). You will almost always find it easier to use AT and TAB in the usual way rather than the control codes, but they might be useful in some circumstances. The AT control character is CHR\$ 22, the first character after it specifies the y-position (but a line number or y-pixel value according to the layer we're currently in, and the second the column number, so that

```
PRINT CHR$ 22+CHR$ 1 +CHR$ c,
```

has exactly the same effect as

```
PRINT AT 1,c,
```

This is so even if CHR\$ 1 or CHR\$c would normally have a different meaning (for instance if c=13) the CHR\$ 22 before them overrides that.

The TAB control character is CHR\$ 23 and the two characters after it are used to give a number between 0 and 65535 specifying the number you would have in a TAB modifier.

```
PRINT CHR$ 23+CHR$ a+CHR$ b,
```

has the same effect as

```
PRINT TAB a+256*b,
```

As with the character size control codes, there are further control codes that only apply to layers other than 0 and further modify their behaviour. One of these is CHR\$ 25 or the *Scroll-prompt inhibitor* control code. Set by CHR\$ 26, CHR\$ n where n is the number of lines that can be scrolled off before the scroll prompt cursor appears (as discussed in the *Scrolling* section above) but after the first full screen length has been printed. If 0 the scroll prompt function is inhibited for that layer/window. Note that the number of lines is calculated based on an 8 pixel character height. That can lead to some very confusing results: your chosen character height is different. Some are easy to calculate like the standard or double height characters, with the latter in essence having the amount of lines but others not so easy as with the reduced height and double reduced height characters. In the worst cases you have to calculate how many pixels your program scrolls vertically by getting the amount of actual lines times the height of the characters and then divide the product by 8 (standard character height) in order to arrive at how many lines you need to instruct the system via the *Scroll-prompt inhibitor* control code to allow.

This sounds unnecessarily complicated that's because it is. In most cases, the average user will either need to disable scroll-prompting by setting n = 0 or just set it to a full screen of data by setting n to 24 (for all screen modes except LAYER 1 0 which requires n set to 12).

On Layer 0 you can duplicate that behaviour albeit in a less confusing way since the characters are always 8 pixels high by employing a bit of POKE trickery to control the scroll prompt by doing


```
POKE 23692,x
```

where *x* is the amount of lines the scroll prompt should be inhibited for (or in other words *every time the scroll counter has been reached*). After this it will scroll up *x* number of lines before stopping again with `scroll`.³ As an example try

```
10 POKE 23692, 255
20 FOR n=1 TO 400
30 PRINT "line ",n
40 NEXT n
```

and watch everything whizz off the screen up until line 277 before the prompt to scroll reappears. The technical explanation of what this **POKE** does, is that it modifies the *System Variable* **SCROLL**. It's important to also note that the Editor resets this *System Variable* so entering the **POKE** directly will have no appreciable effect on scrolling on *Lavaré* until it's entered in a program. We will examine all the possible combinations of **PRINT** control codes on Chapter 2. You will find more information about *System variables* in Chapter 24 and for **POKE** in Chapter 23 *The Memory*.

INKEY\$

There's an additional function related to keyboard entry called **INKEY\$**. **INKEY\$** (which takes no argument) reads the keyboard immediately when it's invoked. You are pressing exactly one key (or a **SHIFT** key and just one other key) then the result is the character that that key gives in that typing mode; otherwise the result is the empty string. Try this program which works like a typewriter:

```
10 IF INKEY$ <>"" THEN GO TO 10
20 IF INKEY$ = "" THEN GO TO 20
30 PRINT INKEY$,
40 GO TO 10
```

Here line 10 waits for you to lift your finger off the keyboard and line 20 waits for you to press a new key.

Unlike the regular **INPUT** (see also the next section) **INKEY\$** doesn't wait for you. So you don't type **ENTER**, but on the other hand if you don't type anything at all then you've missed your chance. This also explains why the **GO TO** statements are needed in lines 10 and 20.

Using INPUT for game controllers

Much like **INKEY\$** above, **INPUT** can also be used as a function with a numeric parameter *n* in order to read the current state of an input controller.

```
INPUT n
```

reads the current state of an input controller which can be one of the two joysticks (if *n* is 1 or 2) or the keyboard joystick⁴ (if *n* is 0).

In each case the value returned is a bitmask of the following value:

bit 0 (value 1)	right pressed
bit 1 (value 2)	left pressed
bit 2 (value 4)	down pressed
bit 3 (value 8)	up pressed
bit 4 (value 16)	fire pressed

³ The ZXOS has a feature where no keyboard can emulate one of the joystick standards & normally supports

bit 5 (value 32)	fire2 pressed
bit 6 (value 64)	fire3 pressed
bit 7 (value 128)	fire4 pressed

For example

```
INPUT 1 & 8      returns false (0) if up is not pressed
                  or joystick 1 true (8 is non-zero) if it is

INPUT 0 & 811110000 returns false (0) if no fire buttons are
                  pressed on joystick 0, true (non-zero) if at least
                  one is
```

The default *keyboard joystick* is set up to use the following keys

up	Q
down	A
left	O
right	P
fire	SPACE
fire2	M
fire3	ENTER
fire4	X

The *keyboard joystick* may be redefined with the INPUT function by if negative values are specified for n as follows

INPUT	1	waits for a key to be pressed and assigns to right
INPUT	2	waits for a key to be pressed and assigns to left
INPUT	3	waits for a key to be pressed and assigns to down
INPUT	4	waits for a key to be pressed and assigns to up
INPUT	5	waits for a key to be pressed and assigns to fire
INPUT	6	waits for a key to be pressed and assigns to fire2
INPUT	7	waits for a key to be pressed and assigns to fire3
INPUT	8	waits for a key to be pressed and assigns to fire4
INPUT	9	(or any other negative value) clears all assignments

The return value is the character code of the key pressed which can be used if you want to display the key just defined although some special keys have codes below ASCII 32 which aren't printable so care should be taken. Here's an example on how to set up the *keyboard joystick*. Note that we cannot use INPUT to print on the screen so separate PRINT statements are needed

```
100 x=INPUT 9
110 ,clear the "keyboard joystick"
120 PRINT 'Press a key for right'
130 x=INPUT 1
140 PRINT "Press a key for left"
150 x=INPUT 2
160 PRINT "Press a key for down"
170 x=INPUT -3
180 PRINT 'Press a key for up'
190 x=INPUT 4
200 PRINT 'Press a key for fire'
210 x=INPUT 5
220 REM Can leave additional fire
    buttons undefined if they aren't
    needed
```


Chapter 15 Colours

An introduction to colour on the ZX Spectrum Next

sors. The first capability which we will examine in depth is colour.

Basics of computer colour

intensities which translates to 512 colours in total.

a colour could be represented in binary form.

Colour organisation and representation

Blue components

later but for now let's assume it can manipulate 3-bit, 8-bit and 9-bit colours.

Spatial vs Colour Resolution

A quick simplification of both how many bits and how many colours we can do. The former is spatial resolution – has one more component, *density* but this is no pertinent to this discussion and the latter, colour resolution, is important to make no distinction as we will see below (see section 16) it's not only a computer's design choices when it comes to graphics but also the special machinery that may be involved to display both on screen.

5 easy to understand spatial resolution. We (as you already read here and probably else where) modes are spatial resolution in pixels – or PICTURE Elements – in essence dots arranged in a Cartesian (with finer spatial coordinate system) leaving colour information aside for the moment we can assign one bit per pixel and we can project this in the computer's memory in a linear fashion. Each horizontal line follows the other so in the end we have a series of bits with each line being w times away from the very first bit that started our picture where w is the horizontal resolution and h is the line we're on. We need $w \times h$ bits to represent the screen spatially where w is as before the horizontal size and h is the vertical size (both of them measured in pixels).

This is very straightforward and indeed the ZX Spectrum Next uses this way of storing graphical data in Layers 2¹ and Layer 0. However in all the older modes, uses a variation of linear storage called interleaved storage. The screen area is separated vertically into three 64 pixel high strips of 8 attribute cells arranged in blocks of 32. A horizontal line x is stored linearly in other words a pixel stored in horizontal coordinate 9 follows the pixel stored in horizontal coordinate 8 however when it comes to the vertical order there is a vertical hops/switch of sorts happening. The computer stores the first line of the first block of attribute cells then stores the first line of the second block until it reaches the first line of the third block then it jumps to the second line of the first block and so on. After it has written all second line of the first block and so on until each third of the screen is full. Fig. 9 demonstrates as the reader is to agree the ZX Spectrum Next legacy modes are more or less a relic. In the better we will get into more detail on why the graphical data is stored in that way later.

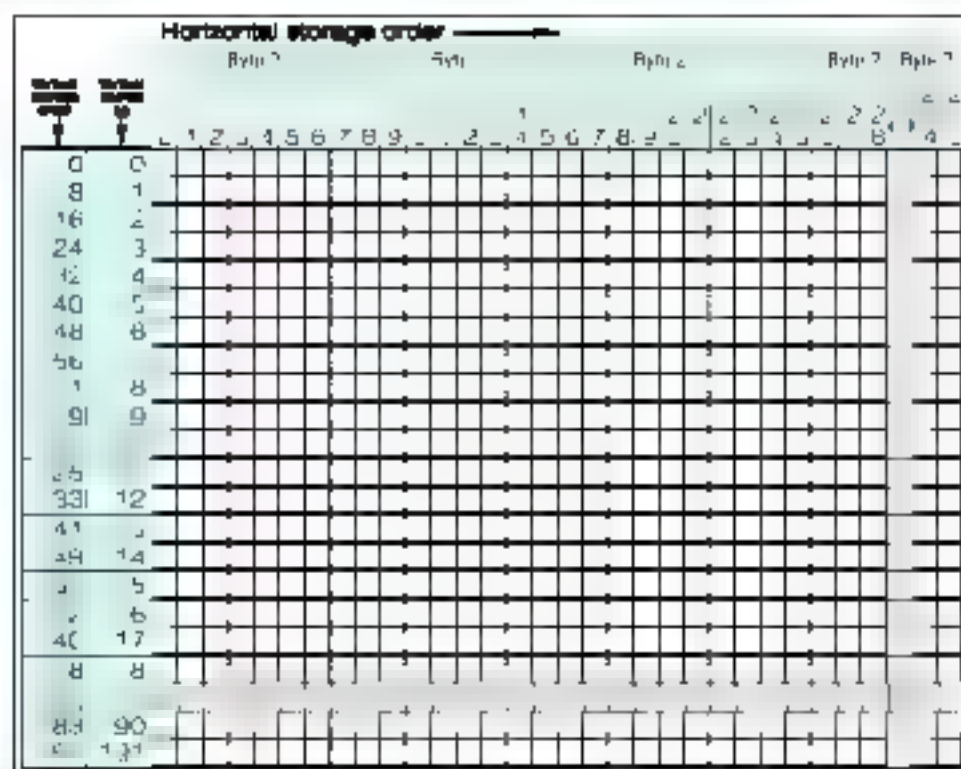


Fig. 9: Interleaved graphical data storage of ZX Spectrum Next legacy modes

¹Layer 0 and Layer 1 resolutions are not currently supported by NextBASIC, store images a bit differently than in the other modes.

g:am

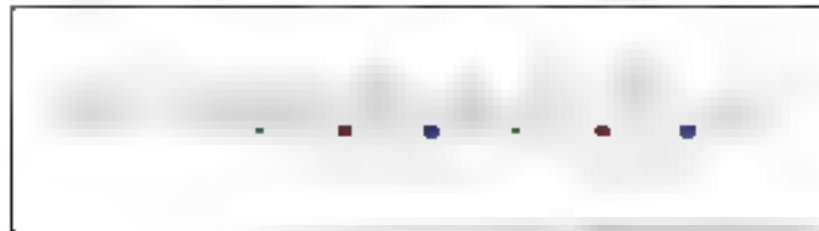
```
10 LAYER 1,2
20 BANK 5 ERASE 0,6144,0
30 FOR %n=0 TO 6143
40 BANK 5 POKE %n,%010101010
50 NEXT %n
```

Memory for more details on the BANK command and its parameters)

letting DISP FILE 1 handle the even ones

```
10 LAYER 1 2
20 BANK 5 ERASE 0,6144,0
30 BANK 5 ERASE 6192 6144,0
40 FOR %n=0 TO 6143
50 BANK 5 POKE %n,%010001000
60 NEXT %n
70 FOR %k=192 TO 6192+6143
```


how colour information is stored in each byte in the COLOUR FILE area



colour resolution is much lower than the *spatial* one

when enabled, is called COL FILE 2 and resides right after DISP FILE 2

also does not use the COLOUR FILE areas but for a different reason

the same interleaved storage as the graphic data

get a visual idea of the two modes' differences

```
10 LAYER 1 1
20 BANK 5 ERASE 0,65535,4
30 BANK 5 ERASE 65536,65535,4
40 FOR %n=0 TO 65535
50 BANK 5 POKE %n,%010101010
60 NEXT %n
```



```

70 FOR %a=6144 TO 6144+767
80 BANK 5 POKE
   %a,INT((RND*1)+0.2)*128 +
   %RND(128)
90 NEXT %a
100 LAYER 1 3
110 FOR %x=6192 TO 6192+6143
120 BANK 5 POKE
   %x,INT((RND*1)+0.2)*128
   + %RND(128)
130 NEXT %x
140 LAYER 1 1
150 PAUSE 8

```

When the attribute bit is set, the colour attribute is randomised independently of the colour value. The program above sets the colour attribute to 3 (blue) and the colour value to a randomised value between 0 and 127. The colour attribute is then set to 1 (red) and the colour value is randomised between 0 and 127. The program then pauses for 8 frames.

Extended colour attribute display

The colour attribute display is a feature of the Z8000 which is not available on the Z8000. The colour attribute display is a feature of the Z8000 which is not available on the Z8000. The colour attribute display is a feature of the Z8000 which is not available on the Z8000. The colour attribute display is a feature of the Z8000 which is not available on the Z8000.

This is achieved by setting the BANK attribute to 0. The BANK attribute is a feature of the Z8000 which is not available on the Z8000. The BANK attribute is a feature of the Z8000 which is not available on the Z8000. The BANK attribute is a feature of the Z8000 which is not available on the Z8000.

```

10 BANK NEW ba
20 FOR %a=0 TO 255
30 BANK ba POKE %a,%a
40 NEXT %a
50 LAYER 1 1
60 PALETTE DIM 0
70 LAYER PALETTE 0 BANK ba, 0

```



```

80 PALETTE FORMAT 255
90 BANK 5 ERASE 0,6912,255
100 %L=6144
110 REPEAT WHILE %L<6912
120 IF %C>255 THEN %C=0
130 BANK 5 POKE %L,%C
140 %L,%C+=%I
150 REPEAT UNTIL 0
170 PAUSE 0

```

Don't worry about the unknown commands yet. What the program does is to read an array of 256 values, and write them to the screen. As the array is filled, the program will keep track of the colour being written to the screen. If the colour is 256 or more, it will just wrap around to 0. The program will stop when it reaches 6912 and the ZX Spectrum Next Ports System in Chapter 22.

Palette-based hybrid linear bitmapped colour display

This system of colour representation is a hybrid of the two systems above. It uses a palette to store the colours, and a linear bitmapped display to store the pixels. The palette is a table of 256 values, each value representing a colour. The linear bitmapped display is a table of pixels, each pixel being a byte that points to a colour in the palette. This system is used in the ZX Spectrum Next and the ZX Spectrum Next Ports System in Chapter 22.

We'll use a small display to illustrate the concept of a palette. A palette is a subarray of the array of 256 values. It is a table of 256 values, each value representing a colour. The linear bitmapped display is a table of pixels, each pixel being a byte that points to a colour in the palette. This system is used in the ZX Spectrum Next and the ZX Spectrum Next Ports System in Chapter 22.

The ZX Spectrum Next Ports System uses the following system:

- Layers 0 and 1 use two
- Layer 2 uses two more
- Layer 3 also uses two - and-
- The Sprite System uses the last two

We use palettes all the time. In the ZX Spectrum Next, we can use a palette to store the colours of the pixels. The palette is a table of 256 values, each value representing a colour. The linear bitmapped display is a table of pixels, each pixel being a byte that points to a colour in the palette. This system is used in the ZX Spectrum Next and the ZX Spectrum Next Ports System in Chapter 22.

The ZX Spectrum Next Ports System uses the following system:

4. The ZX Spectrum Next Ports System uses the following system:

The ZX Spectrum Next Ports System uses the following system:

[illegible]

C:\PROMJIT

[illegible]

Use 5 16 Kbyte banks, requiring a total of 80 Kbytes

1. The first part of the document is a list of names and their corresponding numbers, arranged in two columns. The names are: *John A. Smith*, *John B. Smith*, *John C. Smith*, *John D. Smith*, *John E. Smith*, *John F. Smith*, *John G. Smith*, *John H. Smith*, *John I. Smith*, *John J. Smith*, *John K. Smith*, *John L. Smith*, *John M. Smith*, *John N. Smith*, *John O. Smith*, *John P. Smith*, *John Q. Smith*, *John R. Smith*, *John S. Smith*, *John T. Smith*, *John U. Smith*, *John V. Smith*, *John W. Smith*, *John X. Smith*, *John Y. Smith*, *John Z. Smith*. The numbers are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100.

[illegible]

therefore neither LoRAS nor Layer 2 modes suffer from colour clash

2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 13. 14. 15. 16. 17. 18. 19. 20. 21. 22. 23. 24. 25. 26. 27. 28. 29. 30. 31. 32. 33. 34. 35. 36. 37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50. 51. 52. 53. 54. 55. 56. 57. 58. 59. 60. 61. 62. 63. 64. 65. 66. 67. 68. 69. 70. 71. 72. 73. 74. 75. 76. 77. 78. 79. 80. 81. 82. 83. 84. 85. 86. 87. 88. 89. 90. 91. 92. 93. 94. 95. 96. 97. 98. 99. 100. 101. 102. 103. 104. 105. 106. 107. 108. 109. 110. 111. 112. 113. 114. 115. 116. 117. 118. 119. 120. 121. 122. 123. 124. 125. 126. 127. 128. 129. 130. 131. 132. 133. 134. 135. 136. 137. 138. 139. 140. 141. 142. 143. 144. 145. 146. 147. 148. 149. 150. 151. 152. 153. 154. 155. 156. 157. 158. 159. 160. 161. 162. 163. 164. 165. 166. 167. 168. 169. 170. 171. 172. 173. 174. 175. 176. 177. 178. 179. 180. 181. 182. 183. 184. 185. 186. 187. 188. 189. 190. 191. 192. 193. 194. 195. 196. 197. 198. 199. 200. 201. 202. 203. 204. 205. 206. 207. 208. 209. 210. 211. 212. 213. 214. 215. 216. 217. 218. 219. 220. 221. 222. 223. 224. 225. 226. 227. 228. 229. 230. 231. 232. 233. 234. 235. 236. 237. 238. 239. 240. 241. 242. 243. 244. 245. 246. 247. 248. 249. 250. 251. 252. 253. 254. 255. 256. 257. 258. 259. 260. 261. 262. 263. 264. 265. 266. 267. 268. 269. 270. 271. 272. 273. 274. 275. 276. 277. 278. 279. 280. 281. 282. 283. 284. 285. 286. 287. 288. 289. 290. 291. 292. 293. 294. 295. 296. 297. 298. 299. 300. 301. 302. 303. 304. 305. 306. 307. 308. 309. 310. 311. 312. 313. 314. 315. 316. 317. 318. 319. 320. 321. 322. 323. 324. 325. 326. 327. 328. 329. 330. 331. 332. 333. 334. 335. 336. 337. 338. 339. 340. 341. 342. 343. 344. 345. 346. 347. 348. 349. 350. 351. 352. 353. 354. 355. 356. 357. 358. 359. 360. 361. 362. 363. 364. 365. 366. 367. 368. 369. 370. 371. 372. 373. 374. 375. 376. 377. 378. 379. 380. 381. 382. 383. 384. 385. 386. 387. 388. 389. 390. 391. 392. 393. 394. 395. 396. 397. 398. 399. 400. 401. 402. 403. 404. 405. 406. 407. 408. 409. 410. 411. 412. 413. 414. 415. 416. 417. 418. 419. 420. 421. 422. 423. 424. 425. 426. 427. 428. 429. 430. 431. 432. 433. 434. 435. 436. 437. 438. 439. 440. 441. 442. 443. 444. 445. 446. 447. 448. 449. 450. 451. 452. 453. 454. 455. 456. 457. 458. 459. 460. 461. 462. 463. 464. 465. 466. 467. 468. 469. 470. 471. 472. 473. 474. 475. 476. 477. 478. 479. 480. 481. 482. 483. 484. 485. 486. 487. 488. 489. 490. 491. 492. 493. 494. 495. 496. 497. 498. 499. 500. 501. 502. 503. 504. 505. 506. 507. 508. 509. 510. 511. 512. 513. 514. 515. 516. 517. 518. 519. 520. 521. 522. 523. 524. 525. 526. 527. 528. 529. 530. 531. 532. 533. 534. 535. 536. 537. 538. 539. 540. 541. 542. 543. 544. 545. 546. 547. 548. 549. 550. 551. 552. 553. 554. 555. 556. 557. 558. 559. 560. 561. 562. 563. 564. 565. 566. 567. 568. 569. 570. 571. 572. 573. 574. 575. 576. 577. 578. 579. 580. 581. 582. 583. 584. 585. 586. 587. 588. 589. 590. 591. 592. 593. 594. 595. 596. 597. 598. 599. 600. 601. 602. 603. 604. 605. 606. 607. 608. 609. 610. 611. 612. 613. 614. 615. 616. 617. 618. 619. 620. 621. 622. 623. 624. 625. 626. 627. 628. 629. 630. 631. 632. 633. 634. 635. 636. 637. 638. 639. 640. 641. 642. 643. 644. 645. 646. 647. 648. 649. 650. 651. 652. 653. 654. 655. 656. 657. 658. 659. 660. 661. 662. 663. 664. 665. 666. 667. 668. 669. 670. 671. 672. 673. 674. 675. 676. 677. 678. 679. 680. 681. 682. 683. 684. 685. 686. 687. 688. 689. 690. 691. 692. 693. 694. 695. 696. 697. 698. 699. 700. 701. 702. 703. 704. 705. 706. 707. 708. 709. 710. 711. 712. 713. 714. 715. 716. 717. 718. 719. 720. 721. 722. 723. 724. 725. 726. 727. 728. 729. 730. 731. 732. 733. 734. 735. 736. 737. 738. 739. 740. 741. 742. 743. 744. 745. 746. 747. 748. 749. 750. 751. 752. 753. 754. 755. 756. 757. 758. 759. 760. 761. 762. 763. 764. 765. 766. 767. 768. 769. 770. 771. 772. 773. 774. 775. 776. 777. 778. 779. 780. 781. 782. 783. 784. 785. 786. 787. 788. 789. 790. 791. 792. 793. 794. 795. 796. 797. 798. 799. 800. 801. 802. 803. 804. 805. 806. 807. 808. 809. 810. 811. 812. 813. 814. 815. 816. 817. 818. 819. 820. 821. 822. 823. 824. 825. 826. 827. 828. 829. 830. 831. 832. 833. 834. 835. 836. 837. 838. 839. 840. 84

Layer 3 colour storage

220 1. *Staphylinidae* (1000) 2. *Curculionidae* (1000) 3. *Chrysomelidae* (1000) 4. *Scarabaeidae* (1000) 5. *Elmidae* (1000) 6. *Chrysomelidae* (1000) 7. *Curculionidae* (1000) 8. *Chrysomelidae* (1000) 9. *Curculionidae* (1000) 10. *Chrysomelidae* (1000) 11. *Curculionidae* (1000) 12. *Chrysomelidae* (1000) 13. *Curculionidae* (1000) 14. *Chrysomelidae* (1000) 15. *Curculionidae* (1000) 16. *Chrysomelidae* (1000) 17. *Curculionidae* (1000) 18. *Chrysomelidae* (1000) 19. *Curculionidae* (1000) 20. *Chrysomelidae* (1000) 21. *Curculionidae* (1000) 22. *Chrysomelidae* (1000) 23. *Curculionidae* (1000) 24. *Chrysomelidae* (1000) 25. *Curculionidae* (1000) 26. *Chrysomelidae* (1000) 27. *Curculionidae* (1000) 28. *Chrysomelidae* (1000) 29. *Curculionidae* (1000) 30. *Chrysomelidae* (1000) 31. *Curculionidae* (1000) 32. *Chrysomelidae* (1000) 33. *Curculionidae* (1000) 34. *Chrysomelidae* (1000) 35. *Curculionidae* (1000) 36. *Chrysomelidae* (1000) 37. *Curculionidae* (1000) 38. *Chrysomelidae* (1000) 39. *Curculionidae* (1000) 40. *Chrysomelidae* (1000) 41. *Curculionidae* (1000) 42. *Chrysomelidae* (1000) 43. *Curculionidae* (1000) 44. *Chrysomelidae* (1000) 45. *Curculionidae* (1000) 46. *Chrysomelidae* (1000) 47. *Curculionidae* (1000) 48. *Chrysomelidae* (1000) 49. *Curculionidae* (1000) 50. *Chrysomelidae* (1000) 51. *Curculionidae* (1000) 52. *Chrysomelidae* (1000) 53. *Curculionidae* (1000) 54. *Chrysomelidae* (1000) 55. *Curculionidae* (1000) 56. *Chrysomelidae* (1000) 57. *Curculionidae* (1000) 58. *Chrysomelidae* (1000) 59. *Curculionidae* (1000) 60. *Chrysomelidae* (1000) 61. *Curculionidae* (1000) 62. *Chrysomelidae* (1000) 63. *Curculionidae* (1000) 64. *Chrysomelidae* (1000) 65. *Curculionidae* (1000) 66. *Chrysomelidae* (1000) 67. *Curculionidae* (1000) 68. *Chrysomelidae* (1000) 69. *Curculionidae* (1000) 70. *Chrysomelidae* (1000) 71. *Curculionidae* (1000) 72. *Chrysomelidae* (1000) 73. *Curculionidae* (1000) 74. *Chrysomelidae* (1000) 75. *Curculionidae* (1000) 76. *Chrysomelidae* (1000) 77. *Curculionidae* (1000) 78. *Chrysomelidae* (1000) 79. *Curculionidae* (1000) 80. *Chrysomelidae* (1000) 81. *Curculionidae* (1000) 82. *Chrysomelidae* (1000) 83. *Curculionidae* (1000) 84. *Chrysomelidae* (1000) 85. *Curculionidae* (1000) 86. *Chrysomelidae* (1000) 87. *Curculionidae* (1000) 88. *Chrysomelidae* (1000) 89. *Curculionidae* (1000) 90. *Chrysomelidae* (1000) 91. *Curculionidae* (1000) 92. *Chrysomelidae* (1000) 93. *Curculionidae* (1000) 94. *Chrysomelidae* (1000) 95. *Curculionidae* (1000) 96. *Chrysomelidae* (1000) 97. *Curculionidae* (1000) 98. *Chrysomelidae* (1000) 99. *Curculionidae* (1000) 100. *Chrysomelidae* (1000)

Layer 2 priority colours

[illegible]

■ **By-relocatable:** we mean that although the ZX Spectrum Next initially reserves BANKS 9 through 12 for layer 2 graphics, this can change either automatically or by the user. One should not assume the aforementioned banks of

will display in the LAYER 5 palette. Especially where it involves moving graphics is very processor intensive and can slow down the computer resulting in a noticeable drop in performance. To move the Z80 to the next address this value is stored in the Z register. In the next chapter, we will see how to use the Z register to store the address of the next instruction. These apply only to layer 2 palettes and are defined by setting the 8th bit of the secondary byte of each palette entry to 1.

Setting any palette entry's priority bit will ensure that its colour will always appear for everything else. In case you would need the same colour to exist in a layer below the one you will see it in, then it is safe to do again, but it is different in effect to the LAYER PALETTE command.

We will visit this again further below when we reach the palette manipulation commands.

More on the LAYER command

In Chapter 14 as well as in the previous sections of this book, we saw a brief mention and usage of the LAYER command. By now you should have enough grasp of the mechanics behind the ZX Spectrum Next's colour and graphics system to examine in a bit more detail. We will first expand on the usage of every palette and finally we haven't yet discussed is introduced as in the PALETTE section that follows shortly, but for now let's head back to the beginning of Chapter 14 and reiterate the basic graphics modes in conjunction with LAYER which is used to change between them.

First of all and given what we've learned in terms of colour, it's helpful to conceptualise the graphics system in a slightly different way to what the LAYER command gives them in. These layers are grouped together in terms of functionality and memory address as they are similar. The JLA mode is layer 0 and layer 1 modes are layer 2 and the Sprite system which we will examine more fully in Chapter 17, mode 4. Although it is a potentially confusing as layers 0, 1, 2, and 3 use the same colour storage and display system as they are completely distinct in their operation. In other words, screens 0, 1, 2, and 3 are all independent of one another with their own graphics and potentially graphical effects. In simple words that means that you can select whichever screen you want to display on, which in effect is means putting a priority on the memory space that holds the data of the graphics and displaying this above everything else. This is achieved with the

LAYER OVER order

command, where order is one of the following

- 0 Sprites over Layer 2 over JLA (Layer 1) the default
- 1 Layer 2 over Sprites over JLA (Layer 1)
- 2 Sprites over JLA (Layer 1) over Layer 2
- 3 Layer 2 over JLA (Layer 1) over Sprites
- 4 JLA (Layer 0) over Sprites over Layer 2
- 5 JLA (Layer 1) over Layer 2 over Sprites
- 6 Sprites over (Layer 2 + JLA combined) colours clamped to 7
- 7 Sprites over (Layer 2 + JLA combined) colours clamped to (0, 7)

Effectively, it indicates the order in which the rendering modes allow you to store very interesting lighting/shading effects.

This, as we will see in Chapter 16, the Output to Next Registers, directly affects the Sprite and Layer System Register (Register 2) and is the same value as the LAYER OVER command.

Fig. 19 below visualises the way layers compound to form the ZX Spectrum Next display



Fig. 19 Overlay layers
(Graphics courtesy of Vampiro Polamianos from *The Hollow Earth Hypothesis*)

You will notice a few odd things about the diagram above. First, it is out of order with the sprites appearing below layer 3. This brings us to the second thing I don't worry the docs will be criticised shortly, which is that Layer 0, Layer 1 as well as Layer 3 have a higher basic resolution than Layer 2. The doc was changed to group the like resolutions together and therefore visualise the Layer 3 as well as Layer 0. The system have a maximum of 420 pixels horizontally and 256 pixels vertically resolution as opposed to the 256 pixels by 132 pixels standard resolution of the other layers. As for the doc as seen in the LAYER OVER command, really doesn't make as clear as be arranged the way we see in the specific example above we can see how one can mix-and-match several layers to build a more complex visual. Layer 3 is used for the background, the extended same area for relatively a little information about the game lives and such. Layer 1 for basic parallax animation, clouds and Layer 2 for the remaining more complex and colourful graphics.

It's also noteworthy that although we speak about memory organisation, in regards to colour in all layers, we did not mention the Sprite system. That's because sprites do not occupy normal memory but instead use their own dedicated memory that's isolated within the Next Sprite Engine hardware. The LAYER command and other layer descriptions to display does not affect nor addresses the Sprite Engine directly therefore in the following commands, the latter is not referenced anywhere.

There are more LAYER compound commands that are more pertaining graphics rather than colour and others that deal with motion in some fashion or other. We will revisit therefore LAYER in more detail in the following sections and chapters. The main functionality of the LAYER command which is none other than changing graphic modes.

LAYER number parameter

will change the layer to the one specified by number with an optional parameter according to the list below

- LAYER 0** Select legacy ZX Spectrum Mode
- LAYER 1.0** Select Layer 1 LoRes mode
- LAYER 1.1** Select Layer 1 standard resolution mode
- LAYER 1.2** Select Layer 1 HiRes mode⁸

⁸ This is a legacy feature of the ZX Spectrum 54400, in which memory for Layer 1 was split into high and low resolution modes. This was a hardware limitation. In the Next, the 320x200 standard resolution of the computer screen, which was normally shared by the 160x256 mode, is

LAYER 1,3 Select Layer 1 HiColour mode
LAYER 2 Select Layer 2 mode
LAYER 2,0 Select Layer 2 mode and disable its display
LAYER 2,1 Select Layer 2 mode and enable its display

Attempting to enter a layer number or parameter that's not supported according to this list will result in a **B=integer out of range** error.

There's one more command of note and this is

LAYER CLEAR

which will reset all layer information including banks, mode, the Layer 2 display enable, layer offsets (see Chapter 12) and ordering to defaults. This is also done by **NEW**.

BORDER, PAPER, INK, BRIGHT and FLASH

Run this program:

```

5 LAYER 0
10 FOR m=0 TO 1 BRIGHT m
20 FOR n=1 TO 10
30 FOR c=0 TO 7
40 PAPER c PRINT " ", 4 spaces
50 NEXT c NEXT n NEXT m
60 FOR m=0 TO 1 BRIGHT m PAPER 7
70 FOR c=0 TO 3
80 INK c PRINT c, " ", 3 spaces
90 NEXT c PAPER 0
100 FOR c=4 TO 7
110 INK c PRINT c, " ", 3 spaces
120 NEXT c NEXT m
130 PAPER 7 INK 0 BRIGHT 0

```

This shows the fifteen colours (including white and black and the **BRIGHT** variants) that the ZX Spectrum Next can produce on the screen in switched to Layer 0 (or standard resolution modes of Layer 1) without the EnhancedULA functions enabled. Here is a list of the basic eight (reference: they are also written over the appropriate number keys on your ZX Spectrum Next's keyboard).

| | |
|---|-----------------------|
|  | 0 black |
|  | 1 blue |
|  | 2 red |
|  | 3 purple (or magenta) |
|  | 4 green |
|  | 5 cyan (or pale blue) |
|  | 6 yellow |
|  | 7 white |

If you're thinking to yourself that the total colours (taking account of brightness turned on) should be 16, you'd be technically right; however there cannot be a **BRIGHT** black so the total amount of colours is indeed 15. As you've noticed, the program introduces three commands: **PAPER**, **INK** and **BRIGHT**. If you look back to the *Colour attribute display* section you will recognize the terms immediately. These commands are the primary way of applying colour to objects on screen in *NextBASIC*. There is a number of supporting colour commands as well which will examine further in the following sections.

Before we delve a bit deeper into what each does and how it's very important to understand that the commands operate differently according to the layer we're on and this points back to the different way the ZX Spectrum Next stores colour. When we're dealing with modes that make use of attribute cells we need to think in terms of those cells. **PAPER** here affects the background or, in other words, the place in the cell where graphic data is non-existent (set to 0) whereas **INK** does the exact opposite and affects areas within the same cell where graphic data is existent (set to 1). Moreover these commands affect the entire attribute cell and not just one singular pixel within the cell. In other words, it doesn't matter how many times you set the **INK** or **PAPER** within a particular cell, only the last command will be the one that has the permanent effect for that cell. **BRIGHT** similarly affects the entire cell as we already saw, however it does absolutely nothing. **EnhanceColour** is enabled only if we are on modes that do not support attributes like *HiRes*, *LoRes* and *Layer 2*.

On *LoRes* and *Layer 2*, since attribute cells do not exist, the entire notion of **PAPER** and **INK** should be irrelevant. It's easier, however, to think about colours and how in several ways as the attribute display modes are in terms of a character-based display. Indeed, there's nothing stopping us from having an 8 x 8 character drawn on screen (say a Z) with every single pixel around the character having a different colour, something that's impossible in attribute display modes. This however would be very difficult to do in terms of a singular colour command and for that reason **PAPER** and **INK** commands were simply extended to work in a similar manner as their attribute cell modes counterpart, even where their underlying mechanisms are different. On the other hand, in *HiRes* mode **PAPER** and **INK** commands only serve the purpose of selecting a colour scheme as we will see below. The following table shows all primary colour commands functionality according to the graphics mode we're in.

| | Attribute Modes | | | Non-Attribute Modes | | |
|-----------------------|-----------------|---------|--------------|---------------------|-------|-------|
| | Standard 64k | | Enhanced 64k | | | |
| | Layer 0 | Layer 1 | HiColour | Layer 0 | HiRes | LoRes |
| INK | 0-9*** | 0-7 | | 0-255 | 0-255 | 0-255 |
| PAPER | 0-9*** | 0-7 | | 0-255 | 0-255 | 0-255 |
| BRIGHT | | 0-7 | | 0-7 | 0-7 | 0-7 |
| FLASH | 0-1 8*** | 0-1 | | N/A | N/A | N/A |
| BRIGHT | 0-1 8*** | 0-1 | | N/A | N/A | N/A |
| Palette select | | 0-7 | | N/A | N/A | 0 |

INK: Ink colour and range. Layer 0 and 1 are 0-9 (10 colours) and 0-7 (8 colours) respectively. Enhanced 64k is 0-255 (256 colours).
 PAPER: Paper colour and range. Layer 0 and 1 are 0-9 (10 colours) and 0-7 (8 colours) respectively. Enhanced 64k is 0-255 (256 colours).
 BRIGHT: Brightness and range. Layer 0 and 1 are 0-7 (8 colours) and 0-1 (2 colours) respectively. Enhanced 64k is 0-7 (8 colours).
 FLASH: Flash and range. Layer 0 and 1 are 0-1 (2 colours) and 0-1 (2 colours) respectively. Enhanced 64k is 0-1 (2 colours).
 Palette select: Palette select and range. Layer 0 and 1 are 0-7 (8 colours) and 0-7 (8 colours) respectively. Enhanced 64k is 0 (1 colour).

Table 7: Colour commands functionality according to Graphics Mode/Layer

There is another way of using **INK**, **PAPER** etc. which you will probably find more useful than having them as statements. You can put them as items in a **PRINT** statement followed by `;` and they then do exactly the same as they would have done if they had been used as statements in their own, except that their effect is only temporary (that is as far as the end of the **PRINT** statement that contains them). Thus if you type

```
PRINT PAPER 6, 'x', PRINT "y"
```

then only the x will be on a yellow background.

When used as statements in Layer 0, **INK**, **PAPER**, **BRIGHT** and **FLASH** do not affect the colours of the lower part of the screen where commands and **PRINT** data are typed on. The lower part of the screen uses the colour of the **BORDER** as its **PAPER** colour, value 9 for contrast as its **INK** colour, has **FLASH** turned off and everything is set at normal **BRIGHT**.

BORDER

Incidentally, you have noticed thus far that there is an area you cannot write (normally) to surrounding the area where you can print & draw graphics over. This area is called the **BORDER** and using standard NextBASIC statements you can only change its colour. The statement

BORDER colour

changes the *border* colour to any of the eight normal colours (not B or G) or colours changed by the **PALETTE** statement we shall explore below in length.

INVERSE and OVER

There are two more statements, **INVERSE** and **OVER**, which, when in an attribute mode control not the attributes, but the actual graphic data that is printed on the screen. They use the numbers 0 for off and 1 for on in the same way as **FLASH** and **BRIGHT** do, but those are the only possibilities. If you do **INVERSE 1** then the graphic data printed will be the inverse of their usual form: paper pixels will be replaced by ink pixels and vice versa.

The statement

OVER 1

sets into action a particular sort of overprinting. Normally, when something is written into a character position it completely obliterates what was there before, but now the new character will simply be added in on top of the old one (but see Exercise 1). Note that if the character you're overprinting with has a pixel in the same position with the character you're printing **OVER**, the result will be a blank pixel. In other words, **OVER** is a XOR operation.

This can be particularly useful for writing composite characters, like letters with accents on them, as in this program to print out German letters: 'a' and 'o' with an umlaut above it.

```
10 OVER 1
20 FOR n=1 TO 30
30 PRINT 'a' CHR$ 8 ' ',
40 NEXT n
```

(notice the control character **CHR\$ 8** which backs up one space)

Using colour control codes

The previous example reminded us of the **PRINT** positioning control codes. We can do exactly the same with colours by using the special colour control codes in a similar manner like the one we explored in Chapter 14.

The colour control codes are

```
CHR$ 16 corresponds to INK
CHR$ 17 corresponds to PAPER
CHR$ 18 corresponds to FLASH
CHR$ 19 corresponds to BRIGHT
CHR$ 20 corresponds to INVERSE
CHR$ 21 corresponds to OVER
```

These are each followed by one character that shows a colour by its code, so for instance:

```
PRINT CHR$ 16 + CHR$ 9
```

has the same effect as


```
PRINT INK 0, ...
```

ATTR

The **ATTR** function has the form

ATTR (*line*, *column*)

Its two arguments are the *line* and *column* numbers that you would use in an **AT** statement and its result is a number that shows the colours and so on at the corresponding character position on the screen. You can use this as freely in expressions as you can any other function.

The number that is the result is the sum of four other numbers as follows:

- 128 if the character position is flashing, 0 if it is steady
- 64 if the character position is bright, 0 if it is normal
- 8 *times* the code for the paper colour, and finally-
- the code for the ink colour

For instance, if the character position is flashing and normal with yellow paper and blue ink then the four numbers that we have to add together are $28+0+8*6=48$ and making 177 altogether. Test this with

```
PRINT AT 0,0, FLASH 1, PAPER 6, INK 1,
      , ATTR (0,0)
```

ATTR works only on Layer 0 and that is because it works by reading each COLOUR FILE location. On different modes where the memory organisation and usage differs, will return a number that corresponds to the original COLOUR FILE memory location, which could be for all purposes nonsense. That being said, you can get information on the external colour attribute display, the Enhanced Attributes are what are preserving the screen area hasn't moved. That number will correspond to the indices in use and it changes according to which **PALETTE FORMAT** command is in effect as we'll see below. For the modes it's safer to use the **POINT TO** command which we will examine in Chapter 16.

PALETTE

In previous sections of this chapter we got introduced to the subject of palettes and how they affect colour display and manipulation in each of the colour modes. We also got briefly introduced to the **PALETTE** keyword and a few of its uses. We can now expand a bit more on the subject as **PALETTE** not only affects printing of the characters on screen but also all aspects of graphics including the ZX Spectrum Next's Sprite Engine.

The **PALETTE** keyword can be used as a primary statement or as a modifier to the **LAYER** and **SPRITE** statements to perform a variety of functions that pertain to colour manipulation.

As we saw colour on the ZX Spectrum Next when using extended colour attribute display in any mode that doesn't use attributes, can be defined using 9 bits or 8 bits per colour. The default is 9 when 8 bits are chosen as we have already seen previously for attribute modes. An extended attribute display is applied to the display with a side-effect that only 4 levels of blue are available. In the latter case you can basically ignore all **PALETTE** statements as non-applicable for Layer 2, and this whole section is that matter, however you need to use them if you want to manipulate colours of any of the Layer 1 and Layer 0 sprites and/or change the default colours anywhere in your system or even to recolour an old game or old code that you don't have access to. It affects the way the colour will need to change the indexed colour palette. You can also use the **PALETTE DIM** statement in the form

PALETTE DIM bits

where bits can be 8 or 9

The default colour mode in Layer 1 modes (except LoRes and HiRes) is the standard colour of 256 colours. In order to obtain the extra 256 colours of the Enhanced Colour mode we need to enable the EnhancedCLA functionality. For this you must use the **PALETTE FORMAT** which takes the form

PALETTE FORMAT *ink count*

where *ink count* is a numerical expression specifying the number of inks to be in the palette (1, 2, 5, 9, 16, 25 or 256). When the EnhancedCLA is enabled, **BRIGHT** and **FLASH** are ignored, and **INK** and **PAPER** adopt the appropriate new range of values. Note here that although you can specify **INK** and **PAPER** values up to 256 when writing a colour attribute expression in Layer 1, this will result in a **Invalid Colour** error when the EnhancedCLA is not enabled. To disable the EnhancedCLA functionality you will need to specify an ink count of 0. The standard attributes with 8 inks, 8 papers, bright and flash are then once again supported.

As we saw in Fig. 13 there is an order of display of different layers on screen. Although this is not immediately apparent, this means that it is also possible to make display of the colour more than one graphics layers. That is achieved by assigning a global transparency mask or to a single layer or, in the case of the EnhancedCLA, a transparency index which then colouring the areas of sprites we want to be transparent with the specific colour.

You can set the transparency colour mask or transparency colour index using the following statement:

PALETTE OVER *value*

where *value* is an 8-bit numeric expression which identifies a colour either in RGB24 8-bit format in the case of single graphics layers, the index to the 256 colour value we want to be transparent in the case of the Hybrid layer, or the default global transparency mask and transparency colour index is light magenta (227 = 1110001 in binary).

To reset all palette data and settings to default use the **PALETTE CLEAR** statement.

In the *Palette-based hybrid linear bitmapped colour display* section we first discussed the existence of multiple display layers. It is essential in this case layer 1 is handled in the memory usage paradigm displayed in Fig. 13 so VLA layers get grouped together.

We can switch between palettes using the compound keyword

LAYER PALETTE *n*

where *n* is the palette to use (0 is the current memory usage layer, i.e. if you're in a VLA layer all of it gets affected but not Layer 2 etc).

You can load a palette for the current layer if palette data you have previously stored in memory using the following compound command

LAYER PALETTE *number* **BANK** *bank* *offset*

where *number* is the palette to update (0 or 1 for the current memory usage layer, *bank* is the memory bank number and *offset* is the offset within that memory bank. For more information about **BANK** see Chapter 23 *The Memory*.

Palette data should be either 256 double byte colour entries (each bit in 256 single byte entries) or 8-bit. As we will see, we need to define the palette we need to define the colour format in an RGB24 or 8-bit or RGB24 or 8-bit with every colour component value describing 8 intensities per colour.

In the double byte entry method, the second byte in each sequence only has one bit defined in colour, the third bit in as well as the 8-bit primary with only applies to values

used for Layer 2)². It may seem to be a bit inefficient as it stands, because it appears to be wasting memory but that's only if we store our palette in memory before we load it. Other wise palettes do not use memory at all and they only need to be set once and the memory used by the **BANK** method can be immediately released to the system.

You have already seen an example of this method in the *Extended Colour Attribute Display System* section where a palette is set up first as colours and then assigned into the chosen layer palette. Could you change it to accept double-byte colour values?

The tables that follow show the proper format for single and double-byte palette entries. The integer values are included for a better understanding of the conversion process. In actuality you can use either the **BIN** keyword or the **%@** qualifier to enter binary numbers directly.

| First Byte | | | | | | | | | Second Byte | | | | | | | | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| R ₁ | R ₂ | R ₃ | G ₁ | G ₂ | G ₃ | B ₁ | B ₂ | P ₁ | P ₂ | P ₃ | P ₄ | P ₅ | P ₆ | P ₇ | P ₈ | P ₉ | B ₃ |
| 28 | 64 | 32 | 6 | 8 | 4 | | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 4 | 2 | | 4 | 2 | | 4 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | 0 |

Table 8 Double-byte colour entry

| First Byte | | | | | | | | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|--|
| R ₁ | R ₂ | R ₃ | G ₁ | G ₂ | G ₃ | B ₁ | B ₂ | |
| 28 | 64 | 32 | 16 | 8 | 4 | 2 | | |
| 4 | 2 | | 4 | 2 | | 2 | | |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |

Table 9 Single-byte colour entry

Writing the entire palette into memory is not the only option available to the user in order to program a palette. It is also possible to specify individual colours within the palette using the following compound command (as with the rest of the examples in this section layer here implies a memory space organisational unit).

LAYER PALETTE *number index value*

where *number* is the current layer palette we wish to update (0 or 1), *index* is the index of the palette entry to be updated (0 - 255) and *value* is the colour components value expressed in binary using either the **BIN** keyword or the **%@** qualifier in RGB3 format. That means that the colour in that case is ALWAYS 9-bit. For example

LAYER PALETTE 0,0,BIN 110010011

that sets colour index 0 in palette 0 to a nice pink is exactly the same as

LAYER PALETTE 0,0,%@110010011

Exercises

1. Try:

```
PRINT B CHR$ 8 OVER 1
```

Where the **CHR\$** has cut through the **B**, it has left a white dot. This is the way overprinting works on the ZX Spectrum: two papers or two inks give a paper, one of each gives an ink. This has the interesting property that if you overprint with the same thing twice you get back what you started off with - if you now type

² If the value is 0 (with memory address 0) without colour, may be possible on an HD64 display as the hardware is very capable of displaying colour slowly.


```
PRINT CHR$ 8, OVER 1, /
```

Why do you recover an unblemished B?

2 Type

```
PAPER 0 INK 0
```

Isn't it just as well that these don't affect the lower part of the screen?
Now type

```
BORDER 0
```

and see how well the computer looks after you!
But what will happen if you do the same after giving

```
LAYER 1,3
```

3 Run this program

```
10 POKE 22527+RND*704, RND*127
20 GO TO 10
```

Never mind how this works: it is changing the colours of squares on the screen and the RNDs should ensure that this happens randomly. The diagonal stripes that you eventually see are a manifestation of the hidden pattern in RND — the pattern that makes it *pseudorandom* instead of truly random.

4 Type in the chess piece characters in Chapter 13 and then type in this program which draws a diagram of chess positions using them

```
5 REM draw blank board
10 bb,bw=1,2 REM red and
  blue for board
15 PAPER bw INK bb CLS
20 PLOT 79,128 REM border
30 DRAW 65,0 DRAW 0, 65
40 DRAW 65,0 DRAW 0,65
50 PAPER bb
60 REM board
70 FOR n=0 TO 3 FOR m=0 TO 3
80 PRINT AT 6+2*n, 11+2*m,
90 PRINT AT 7+2*n, 10+2*m, ' '
100 NEXT m NEXT n
110 PAPER 0
120 pw,pb=6,5 REM colours of
  white and black pieces
200 DIM b$(8,8) REM positions of pieces
205 REM set up initial positions
210 b$(1)='rnbqkbr'
220 b$(2)= 'pppppppp'
230 b$(7)= 'pppppppp'
```



```

240 b$ 8 = 'ANBQKBNR '
300 REM display board
310 FOR n=1 TO 8   FOR m=1 TO 8
320 bc=CODE b$(n,m)  INK pw
325 IF bc=CODE ' ' THEN GO TO 350
      REM space
330 IF bc>CODE 'Z' THEN INK pb
      bc =32 , lowercase for
      back
340 bc+=79 , convert to
      graphics
350 PRINT AT 5+n, 9+m  CHR$ bc
360 NEXT m   NEXT n
400 PAPER 7   INK 0

```

- 5 The program in p. 86 has a non-apparent flaw. Can you improve on it so it becomes faster?
- 6 Write a version of **ATTR** using a **PROCedure** that will work always, no matter the mode. You can peek ahead if you so wish!
- 7 Using the global transparency colour, palettes and layers can you write a program that will display **All 512 colours** of the ZX Spectrum Next on screen? (It's easier than you think)

Chapter 16 Graphics

In this chapter, we shall see how to draw pictures on your ZX Spectrum Next's screen. As we learned in Chapters 14 and 15, Layer 0 can only use 128 pixels out of its maximum 192 pixel vertical resolution while the other layers accept the maximum height defined by the layer as their vertical resolution. Moreover, if you recall Fig. 9 and 10, Layer 0 has a different graphics coordinate origin from the rest of the layers/modes, located at the bottom leftmost of the screen instead of the top leftmost. All basic graphics commands that we will explore (**PLOT**, **DRAW**, **CIRCLE** and **POINT**) accept both coordinate origins while the **LAYER** and **TILE** commands (as well as the **\$SPRITE** command we'll explore in the following chapter) accept only the top leftmost corner as the coordinate origin. The side-effect of these inverted coordinate systems is that most graphics you will program will appear inverted on the y-axis if you do not account for that difference. We'll illustrate this fact shortly.

PLOT

The statement

PLOT *x coordinate, y coordinate*

puts a dot in the pixel with these coordinates, so this measily program

```
10 PLOT INT (RND*128), INT
   (RND*96) INPUT a$: GO TO 10
```

plots a random point each time you press **ENTER**. This will work on all layers, although it will not use the entire area of the screen in all modes. Can you figure out why?

Here is a rather more interesting program. It plots a graph of the function \sin a sine wave, for values between 0 and 2π .

```
10 FOR n=0 TO 255 REM change
   to 127 for LoRes
20 PLOT n,80+80*SIN (n/128*PI)
30 NEXT n
```

This next program plots a graph of \sqrt{x} (part of a parabola) between 0 and 4.

```
10 FOR n=0 TO 255
20 PLOT n,80+50*(n/64)
30 NEXT n
```

Notice that when in Layer 0, pixel coordinates are rather different from the line and column in an AT item. You may find the programs in Chapter 14 useful when working out pixel coordinates and line and column numbers for Layer 0. The other layers, as we've already discussed, are pretty straightforward. To illustrate, switch to *Hires* and try again. What you see when entering

```
5 LAYER 1,2
10 FOR n=0 TO 255
20 PLOT n,80+50*(n/64)
30 NEXT n
```

and run the program is exactly what we were talking about earlier. Our graph has changed both orientation and stops at the middle of the screen's width. To make the full

All layers, EXCEPT Layer 7 and the High Res versions of Layer 2, as they're not directly supported by NextBASIC.

put similar to the the first iteration of the program you will need to change the **FOR** loop and **PLOT** commands to

```
10 FOR n = 0 TO 511
20 PLOT n,80+50R((511-n)/128)
```

This will invert the coordinates to simulate the Layer 0 display by drawing inverted, extend the **PLOT** x coordinate to 512 pixels and make sure the **PLOT** doesn't get out of bounds (that's why we divide by 128 instead of 64) in reality you do not need to check if you **PLOT** out of bounds for layers other than Layer 0, as graphics commands for these accept locations outside the screen's pixel boundaries, however it's good practice to do so if you want your program to work across layers.

DRAW and CIRCLE

To help you with your pictures, the computer will draw straight lines, circles and parts of circles for you, using the **DRAW** and **CIRCLE** statements.

The statement **DRAW** to draw a straight line takes the form

DRAW *x coordinate* *y coordinate*

The starting place of the line is the pixel where the last **PLOT**, **DRAW** or **CIRCLE** statement left off (this is called the **PLOT** position). **RUN**, **CLEAR**, **CLS** and **NEW** reset it to the coordinate 0 of the selected Layer, bottom left hand corner, at (0,0) for Layer 0, top left hand corner for all other layers, and the finishing place is *x* pixels to the **RIGHT** of that and *y* pixels **UP** or **DOWN**, depending on which layer you're on. This would be **UP** for Layer 0 and **DOWN** for all other layers. The **DRAW** statement on its own determines the length and direction of the line, but not its starting point.

Experiment with a few **PLOT** and **DRAW** commands, for instance

```
PLOT 0,100 DRAW 80, 35
PLOT 90,150 DRAW 80, 35
```

Notice that the numbers in a **DRAW** statement can be negative, although those in a **PLOT** statement can't. Remember always, that the display direction of the **DRAW** statement changes according to the coordinate system used, ergo which layer you choose is very important. You can also plot and draw in colour, although you have to bear in mind all that was discussed in Chapter 15. Depending on the chosen layer, colours may cover the whole of an attribute position instead of individual pixels. Only L0Res and Layer 2 modes offer full individual colour pixel control whereas other layers rely on the attribute used. The following program demonstrates this.

```
10 LAYER 2,0 REM disable Layer
   2
20 FOR n=0 TO 5
30 PROC LayChange (n)
40 BORDER 0 PAPER 0 INK 7 CLS
   REM black out screen
50 x1,y1=0 REM line start
60 c=1 REM ink, starts with
   blue
70 FOR r = 0 TO 9 REM 10
   repetitions
```



```

60  x2=INT (RND*256)  y2=INT (RND*128)
    REM random line end
80  DRAW INK c,x2 x1,y2 y1
100 x1,y1=x2 y2  REM next line starts
    where last one finished
110 c+=1. IF c=8 THEN c=1
120 NEXT r
130 PAUSE 0  REM Display inspection
140 NEXT n
150 STOP
1000 DEFPROC LayChange mode)
1010 IF mode=0 THEN LAYER 0
1020 IF mode=1 THEN LAYER 1,0
1030 IF mode=2 THEN LAYER 1,1
1040 IF mode=3 THEN LAYER 1,2
1050 IF mode=4 THEN LAYER 1,3
1060 IF mode=5 THEN LAYER 2,1
1070 ENDPROC

```

n layers other than LoRes and Layer 2, you can see how the lines seem to get broader as the program goes on, and this is because a line changes the colours of all the pixels in pixels in all the attribute rings it passes through. You may also be temporarily perplexed about how the program doesn't crash on LoRes given that the selected values can exceed those in the physical resolution (see line 81). This would definitely be true, for compatibility reasons on Layer 0, however, on other layers, graphics output off screen is permitted for x and y values up to 65535. Note that you can embed **PAPER INK FLASH** only on layers that this is available or not turned off by enabling the *Enhanced 4 A* functionality. **BRIGHT**, **dim**, **INVERSE** and **OVER** items in a **PLOT** or **DRAW** statement just as you could with **PRINT** and **INPUT**. They go between the keyword and the coordinates, and are terminated by either semicolons or commas.

An extra frill with **DRAW** is that you can use it to draw parts of circles instead of straight lines by using an extra number to specify an angle to be turned through before the line is

DRAW x coordinate y coordinate arc turn

x coordinate and y coordinate are used to specify the finishing point of the line just as before and arc turn is the number of radians that it must turn through as it goes: if arc turn is a positive it turns to the left while if arc turn is a negative it turns to the right. Another way of seeing arc turn is as showing the fraction of a complete circle that will be drawn: a complete circle is 2π radians, so if $a=\pi$ it will draw a semicircle; if $a=0.5*\pi$ a quarter of a circle and so on.

For instance suppose $a=\pi$. Then whatever values x and y take a semicircle will be drawn. Run,

```
10 PLOT 100,100  DRAW 50 50, PI
```


which will draw this:



Fig. 14 Arc drawn with *DRAW* statement

When run on Layer 0, the drawing starts off in a south-easterly direction, but by the time it stops it's going north-west: in between it has turned round through 180 degrees or π radians, the value of π . Obviously, when run in other layers, the vertical part of the drawing is inverted in line with everything we have discussed.

Run the program several times, with π replaced by various other expressions e.g. $-\pi$, $\pi/2$, $3*\pi/2$, $\pi/4$, $1/0$.



The last statement in this section is the **CIRCLE** statement, which draws an entire circle. You specify the coordinates of the centre and the radius of the circle using

CIRCLE *x coordinate* *y coordinate* *radius*

Just as with **PLOT** and **DRAW**, you can put the various sorts of colour terms in at the beginning of a **CIRCLE** statement. As with its **PLOT** and **DRAW** counterparts, **CIRCLE** when used in Layer 0 will produce an error for circles drawn out of bounds but the remaining layers will happily draw off-screen.

POINT, POINT TO

The **POINT** function informs you of the contents of a pixel on screen. It accepts two parameters enclosed in parentheses: *x coordinate* and *y coordinate*. **POINT** in its own works only on Layer 0 and returns 1 if the pixel is set or 0 if not set. Whilst in Layer 0, try

```
CLS PRINT POINT (0,0) PLOT 0,0
PRINT POINT (0,0)
```

There's an extended version of **POINT** utilising the **TO** modifier which works on *all* layers that takes the output of **POINT** and stores it in variable *var*. This returns 1 if the pixel is set or 0 if not set in all layers except *LoRes* and *Layer 2* just as the main **POINT** does in *LoRes* and *Layer 2*; however, it returns a value from 0 to 255 which is the actual palette index entry.

that the pixel with these coordinates is set to 0. To illustrate this rewrite the previous example as

```
CLS POINT 0,0 TO 1 PRINT 1 PLOT
0 0 POINT 0,0 TO 1, PRINT 1
```

Although this may not be the best example for the benefits of using **POINT TO** instead of the simple **POINT** you can save a lot of typing by foregoing a lot of **LET** statements whilst at the same time making your code a lot easier to read and working in every graphics mode. It's important to mention that **POINT TO** does not return the contents of a sprite that's currently on the given coordinates on screen and instead will return the contents of the layer it's run on.

NOTES

POINT here is a function and not a **PRINT** modifier. Note the distinction as it's important.

Using OVER and INVERSE with graphics commands

Enter screen mode (**EDIT** for *NextBASIC Menu* and then the *Screen* option; in the editor and then type

```
PAPER 7 INK 0
```

and let us investigate how **INVERSE** and **OVER** work inside a standard graphics statement. These two affect just the relevant pixel, and not the rest of the character positions. They are normally off (0) in a graphics statement, so you only need to mention them in turn them on: 1.

Here is a list of the possibilities for reference:

- **PLOT** This is the usual form. It plots an ink dot. It sets the pixel to show the ink colour.
- **PLOT INVERSE 1** This plots a dot of ink on a calculator. It sets the pixel to show the paper colour.
- **PLOT OVER 1** This changes the pixel over from whatever it was before, so if it was ink colour it becomes paper colour, and vice versa.
- **PLOT INVERSE 1 OVER 1** This leaves the pixel exactly as it was before, but note that it also changes the **PLOT** position, so you might use it simply to do that.

As another example of using the **OVER** statement fill the screen up with writing using black on white, and then type

```
PLOT 0,0 DRAW OVER 1,255,175
```

This will draw a fairly decent line, even though it has gaps in it wherever there's some writing. Now type exactly the same command again. The line will vanish without leaving any traces whatsoever. This is the great advantage of **OVER 1**. If you had drawn the line using

```
PLOT 0,0 DRAW 255,175
```

and erased it using

```
PLOT 0,0 DRAW INVERSE 1 255,175
```

then you would also have erased some of the writing. Now try

```
PLOT 0,0 DRAW OVER 1,250,175
```


and try to undraw it by

```
DRAW OVER 1, 250, 175
```

This doesn't quite work, because the pixels the line uses on the way back are not quite the same as the ones that it used on the way down. You must undraw a line in exactly the same direction as you drew it.

Note that being in *graphics mode* in the editor is required for the examples above, otherwise the screen will be reset after each command and you will not get to see the results of the **OVER** and **INVERSE** modifiers.

Using stippling patterns to generate additional colours

One way to get unusual colours is to mix two normal ones together in a single square using a user-defined graphic. These patterns are called *stipples* and work reasonably well in lower layers other than LoRes (where the pixels are 'too big') and exceptionally well in Layer 2 where both the available colours and resolution combine to make the results quite believable. Run this program:

```
1000 FOR n=0 TO 6 STEP 2  
1010 POKE LSR 'a'+n, BIN  
      01010101 POKE LSR  
      'a'+n+1, BIN 10101010  
1020 NEXT n
```

which gives the user-defined graphic corresponding to a chessboard pattern. If you print this character (Graphics mode then **A**, in red ink on yellow paper, you will find it gives a reasonably acceptable orange. You can obviously simulate the same behaviour with **PLOT** statements. This is slower than JOGs but it's much more flexible in the diversity of patterns that you can create.

Quick erase and fill using LAYER ERASE

NextBASIC lacks a dedicated fill command, however large rectangular areas on screen can be filled (or emptied) in LoRes and Layer 2 using the compound **LAYER ERASE** statement with 4 coordinate parameters (+ 1 optional fill parameter). The command

```
LAYER ERASE x1 y1,x2,y2 c
```

will fill the rectangular area delineated by (x1 y1) and (x2 y2) with the global transparency colour (if the optional c parameter is not specified) or with the colour index contained in the c parameter taken from the active palette for the selected layer.

Clipping windows

One of the nicer features that come as a result of the layer system is the ability to superimpose, combine graphics that exist in separate memory spaces. This is possible on the one hand due to the existence of the transparency colour and on the other hand due to the ability to order the layer superimposition order. The latter is controllable via the **LAYER OVER** compound command as we saw in the *More on the LAYER command* section in Chapter 14.

This can be further enhanced with the creation of clipping windows which are basically smaller areas of a certain layer where all display in this layer goes and leaves the layers underneath visible (without having to set the entire area to be visible to a transparent colour). If you wish to visualise this, imagine a glass window with a rectangular section painted so you cannot see what's behind. That rectangular section is the clipping window. In essence the opposite of a regular window. The compound command

LAYER DIM *x1 y1 x2 y2*

sets the clip window for the current layer from *x1 y1* to *x2 y2*. Areas of the layer outside this window will be visible. Note that *x1* and *y1* includes the clip window layer's layer data, the sprite system have their own selected clip windows, see the chapter on multi-forms and how kpp's windows are created using the Vexi Registers. The compound command

LAYER CLEAR

will reset all layer information to defaults. This is also done by **NEW** if these banks mode *layer 2* enables all's layer effects including windows and layer ordering.

Tiling

in the right graphics, tiling is an essential technique. If you take a set of images that are all the same data or are the same and tiles screens very quickly something that can be very useful, especially when it is a method that is a bit complicated. These parameters are tiled and then the other parameters are tiled, they are a self-contained graphic for any pattern. Tiles can be repeated as many times as we need them to, or be completely independent.

Each tile can be 8x8 pixels or 16x16 pixels in size. This allows a 64K bank to hold 256 8x8 tiles, 64 16x16 tiles, or 16 32x32 tiles. The bank can hold a complete set of 4096 tiles, or a single 64K bank and a complete set of 64K tiles, or a complete set of 64K banks. You use **EXT** to set the address of the bank to be used and then introduce the memory to the tiles. *x1 y1 x2 y2* is the address of the first tile in the bank. *tilesize* is the size of the tile. And finally, *offset* is the offset of the tile in the bank. A bank of 64K tiles can hold 256 8x8 tiles or 64 16x16 tiles or 16 32x32 tiles. A bank of 64K tiles can hold 256 8x8 tiles or 64 16x16 tiles or 16 32x32 tiles. A bank of 64K tiles can hold 256 8x8 tiles or 64 16x16 tiles or 16 32x32 tiles.

The **tilemap** can be tiled to contain a single 64K bank. This gives a maximum **tilemap** size of 256x64, 128x128, 2048x8 etc.

Any pixels in a tile which are the same colour as the global transparency colour will not be written to the screen, you want to draw pixels, drawing the global transparency colour, you can temporarily change the colour to the global transparency colour using the **PALETTE OVER** command. The **TILE** command will use the **LAYER ERASE** command. See the **Global erase and** section above. To clear regions of the screen to the global transparency colour before drawing tiles on top.

Layer 2 and 3 tiles are stored separately, so you can use both simultaneously. The **TILE** commands affect the currently selected layer/mode. These are

TILE BANK *n*

which defines bank *n* as containing the tiles *tilesize* banks *n* is 0-3 if *tilesize* is

TILE DIM *n offset w tilesize*

defines bank *n* as containing the **tilemap** starting at *offset* in the bank. The **tilemap** is width *w* (1-2048) and uses 8x8 (*tilesize*=8) or 16x16 (*tilesize*=16) tiles.

TILE**TILE AT *x,y***

Draws a tile to the screen from **tilemap** from the offset *x,y* in the **tilemap** to the screen.

TILE *w,h***TILE *w,h* AT *x,y***

```

    bank = 0;    tilesize = 8;    offset = 0;    w = 1;    h = 1;    x = 0;    y = 0;
    bank = 1;    tilesize = 16;   offset = 0;   w = 2;   h = 2;   x = 0;   y = 0;
    bank = 2;    tilesize = 8;    offset = 0;    w = 1;    h = 1;    x = 0;    y = 0;
    bank = 3;    tilesize = 16;   offset = 0;   w = 2;   h = 2;   x = 0;   y = 0;

```


TILE *w,h* **TO** *x2,y2*
TILE *w,h* **AT** *x,y* **TO** *x2,y2*

The above draw a section of screen from a *tile/map*. Number of tiles to draw is width *w* height *h*. The **AT** draws from tile offset *x,y* in the *tilemap* (or 0,0 if not specified as in the previous example) and the **TO** draws to the tile offset *x2,y2* on the screen (or 0,0 if not specified).

Accessing non-supported graphics modes

We spoke previously about how NextBASIC does not support the higher resolutions provided by Layer 2 and the Next hardware. That's not entirely accurate however as there are ways through the power of Next Registers (see Chapter 22) and **BANK POKE** (see Chapter 23) to do it regardless. There are a few unknown commands in the following short program which you will soon encounter, however, the point of the exercise is to show what is possible. For now type it and run it and we'll revisit it in Chapter 23:

```
10 RUN AT 3
20 REG 112, 010000 REM L2
   SELECT 320x200
30 REG 24, 0 REG 24, 159 REG
   24, 0 REG 24, 255 REM
   setup clipping window
40 REG 105, 128 REM SHOW L2
50 FOR %f=0 to 16383
60 BANK 9 POKE %f, %AND(255)
70 BANK 10 POKE %f, 30
80 BANK 11 POKE %f, %AND
   (230)
90 BANK 12 POKE %f, 60
100 BANK 13 POKE %f, %AND
   (128)
110 NEXT %f
120 PAUSE 0
```

Exercises

- 1 Play about with **PAPER**, **INK**, **FLASH** and **BRIGHT** items in a **PLOT** statement. These are the parts that affect the whole of the character position containing the pixel. Normally it is as though the **PLOT** statement had started off
PLOT PAPER 8, FLASH 8, BRIGHT 8
 and only the ink colour of a character position is altered when something is plotted there, but you can change this if you want. Be especially careful when using colours with **INVERSE 1** because this sets the pixel to show the paper colour, but changes the ink colour and this might not be what you expect.
- 2 Try
CIRCLE 100, 57, 50, DRAW 50, 50
 You can see from this that the **CIRCLE** statement leaves the **PLOT** position at a rather indeterminate place – it is always somewhere about halfway up the right hand side of the circle. You will usually need to follow the **CIRCLE** statement with a **PLOT** statement before you do any more drawing.

Chapter 17 Time and Motion

One of the most important features of the ZX Spectrum Next is the ability to move things on screen as either via the usage of sprites or by quickly marching and filling screens to create animations and general visual effects. Motion and animation, however, as in real life, is a function of time. In other words we need to precisely count time in order to display things and for this purpose this chapter will deal with these two seemingly unrelated subjects in one unit. We will begin with the whole idea of timekeeping on the computer and all the facilities the ZX Spectrum Next has in order for us to measure time.

Timekeeping is essential in computing as all devices work on the basis of a unit of time (in our case μs or ns), but much of this happens behind the scenes. Here we will examine the commands related to time together with the optional timing hardware, before we move into animation, scrolling, the Sprite Engine and eventually to the Copper.

PAUSE

While the general attitude in programming is to make things execute as fast as possible, we often find ourselves in need of making our program wait for a specific length of time or even indefinitely. There is a number of reasons why that would be the case: expecting user interaction is one, displaying warnings is another, timing precisely something is a third and, or all the above and more, you will find the **PAUSE** statement useful.

PAUSE n

stops computing and displays the picture for *n* frames of the selected display mode.

In 50Hz mode there are 50 frames-per-sec and 1fps, so setting *n* to 50 would result in 1 sec. pause. Respectively in 60Hz mode which runs at 60fps, this figure would be 60 for 1 sec. pause.

These modes are set there at the Configuration boot menu or via the config file which is located in the `c:\machines\next_` folder. Generally speaking, almost all modern HDMI and VGA displays operate at 60Hz, while many also have 50Hz modes.

n can be up to 65535, which gives you just a little over 21 minutes at 50Hz and just under 19 minutes at 60Hz respectively. If *n* is set to 0, then it means **PAUSE** indefinitely.

A pause of any length, including the indefinite ones, can always be cut short by pressing a key. Note that **CAPS SHIFT + Space** or **BREAK** will cause a break as well. You have to press the key down after the pause has started.

This program works the second hand of a clock:

```
10 REM We select the appropriate
   pause
20 wait=52 REM 50Hz/50=1 sec.
30 REM First we draw the clock face
40 FOR n=1 TO 12
50 PRINT AT 10+10*COS(n/6*PI),
   15+10*SIN(n/6*PI),n
60 NEXT n
70 REM Now we start the clock
80 FOR t=0 TO 200000 REM t is the
   time in seconds
90 a=t/30*PI : REM a is the angle of
   the second hand in rad.
100 sx,sy=60*SIN a,60*COS a
200 PLOT 128,88 DRAW OVER 1,
   sx,sy REM draw 2nd hand
```



```

210 PAUSE wait
220 PLOT 128,68 DRAW OVER 1,
    sx,sy. REM erase 2nd hand
400 NEXT t

```

This clock will run down after about 55.5 hours because of line 60, but you can easily make it run longer. Note how the timing is controlled by line 20. When running in 50Hz mode, you might expect PAUSE 50 to make a tick once a second, but the computing takes a bit of time as well and has to be allowed for. This is best done by trial and error, timing the computer clock against a real one, and adjusting line 20 until they agree. (You can't do this very accurately: an adjustment of one frame in one second is 1.67% or less, or an half an hour in a day.)

Using POKE and PEEK at the System Variables

There is a much more accurate way of measuring time. This uses the contents of certain memory locations. The data stored is retrieved by using PEEK. Chapter 24, *The System Variables*, explains what we're looking at in detail. The expression used is

```
(65536*PEEK 23674 + 256*PEEK 23673 + PEEK 23672)/50
```

which gives the number of seconds since the computer was turned on (up to about 3 days and 21 hours, when it goes back to 0). That being said, System variables are not guaranteed to be in the same location or even accessible with successive versions of NextBASIC and NextZXOS. For that reason, we are provided with one hybrid command/function which does the exact same thing removing unnecessary calculations and non-standard access to the system variables. The keyword in question is TIME, and we'll examine it below.

TIME (command/function)

When used as a command, TIME takes no arguments and simply resets the frame counter, System Variable FRAMES. See Chapter 24 for details. It's intended for use with the corresponding TIME function which returns how many frames have passed since bootup or since the FRAMES counter was reset. For example, here's a silly program to verify PAUSE and TIME measure the same thing:

```

10 INPUT "How many PAUSE
    frames? ",f
20 TIME
30 PROC perftest(f)
40 PRINT "perftest : took
    ",TIME," frames against
    desired ",f," frames."
50 STOP
60 DEFPROC perftest,(frames)
70 PAUSE frames
80 ENDPROC

```

Retrieving information from the RTC

Your ZX Spectrum Next has a JS-307 *Real Time Clock (RTC)* installed, which allows you to use a more accurate way of retrieving timekeeping data, one that doesn't involve any calculations as described above, nor one that can be affected by clock speed changes.

There are two ways to retrieve time (or date) information from the RTC. The first is not very straightforward owing to the fact that it's triggered via a *dot* command. The second how

ever is via *NextBASIC* and it's the aptly named function **TIMES**. Let's examine both as the first method can be used for several other *NextZXOS* facilities that return information for which *NextBASIC* doesn't yet have a specialised keyword.

We need to use the *NextZXOS* facilities of *Channels* and *Streams*, which we will explore in *Chapter 20* and specifically *Channel.v* which opens a *stream* to a fixed sized variable *t\$*:

```

DIM t$(100) OPEN #2,'v>t$' .TIME
CLOSE #2 PRINT t$

```

then by string slicing *t\$* as seen in depth in *Chapter 7* we can extract the information we need to use (time or date) in our programs.

TIMES

Function **TIMES** does the exact thing as the example above but in a much cleaner way and moreover the user does not need to get only time or date separately as with **TIMES** you get them both simultaneously.

For example typing

```
PRINT TIMES
```

will return a string of the format *yyyy-mm-dd hh-mm-ss* like

```
2023 05 10 16 55 32
```

which is obviously easy to slice and get the information from.

Now we already discussed that a frame lasts a different amount of time depending on if your computer is running at 50Hz or at 60Hz. Here's a revised program that finds out what frequency you're running on first and then tests against the amount of **PAUSE** frames you'll request. Do not pay attention to the **REG** function, we will explain that *Chapter 22*.

```

10 sp=REG 5&0100>>2
20 PRINT 'I'm running at
   ,sp? '50 '60 ',Hz
30 INPUT 'How many PAUSE
   frames? ',f
40 PRINT
50 PROC gettime() TO H1,M1,S1
60 PRINT 'Testing at
   ,sp? '50 '60 ') Hz
   started
   at ,H1 ',M1, ',S1
70 TIME
80 PROC PerfTest(f)
90 PRINT 'PerfTest ) took
   ',TIME," frames against
   desired ',f," PAUSE
   frames '
100 PROC GetTime() TO H2, M2,
   S2

```



```

110 PRINT 'Testing at
    ',sp?('50','60'), 'Hz
    ended
    at '.H2' ',M2,' 'S2
120 STOP
130 DEFPROC perftest (frames)
140 PAUSE frames
150 ENDPROC
160 DEFPROC GetTime
170 a$=TIME$
180 h,m,s=VAL(a$(12 TO 13)),
    VAL(a$ 15 TO
    16)),VAL(a$(18 TO),
190 ENDPROC = h,m,s

```

Line 0 reads the *Nexi Register bit* that holds the vertical frequency to see if it's 50 or 60Hz before asking you how many **PAUSE** frames you want to test again. The next thing that happens is that procedure **GetTime** is called which gets the current time in a string formatted as described above in the section devoted to **TIME\$**. By using string slicing and the **VAL** function we can get the hour, minute and second separately in case we need to manipulate or use them later. Immediately afterwards the **TIME** command is initiated which as we mentioned above resets the frame counter. Following that we call procedure **PerTest** which performs a **PAUSE** for as many frames as we've asked already for and then we inform the user of the requested **PAUSE** frames and the actual frames as returned by function **TIME** match. Finally we get once again the current time, store it in variables **h2 m2** and **s2** and display it. If you run the program for enough frames and you perform some arithmetic using **s1 m1 h1** and **s2 m2** and **h2** you'll easily figure out how much time a frame as requested by **PAUSE** lasts.

Here is a revised clock program to make use of this

```

10 REM First we draw the clock face
20 FOR n=1 TO 12
30 PRINT AT 10+10*COS(n/6*PI),
    15+10*SIN(n/6*PI),n
40 NEXT n
50 DEF FN secs()=VAL TIME$(18 TO)
    REM Get the number of seconds
100 REM Now we start the clock
110 t1=FN secs()
120 a=t1/30*PI REM a is the angle of
    the second hand in radians
130 sx,sy=72*SIN a,72*COS a
140 PLOT 131,91 DRAW OVER 1,sx,sy
    REM draw hand
200 t=FN secs()
210 IF t<=t1 AND t<>0 THEN GO TO 200
    ELSE IF t=0 and t1 > t then go to
    220 else go to 220 REM wait
    until time for next hand except
    if seconds reset to 0

```



```

220 PLOT 131,81 DRAW OVER 1,5X 5Y
    REM rub out old hand
230 t1=t GO TO 120

```

The real time clock that this method uses should be accurate to about 001% regardless if the computer is just running its program or about 1 second per day. Unlike the PEEK method shown above where the computer (and the courier) stops temporarily whenever you get a BEEP or a carriage device operation or use the printer or any of the other extra pieces of equipment you can use with the computer. All these would make it lose time however using the RTC which runs independently bypasses this issue.

If you have chosen to run your ZX Spectrum Next in 60Hz mode then for any program that uses PAUSE you must replace 50 by 60 where appropriate.

INKEY\$

The function INKEY\$ (which has no argument) reads the keyboard. If you are pressing exactly one key (or a SHIFT key and just one other key) then the result is the character that that key gives in L mode, otherwise the result is the empty string.

Try this program which works like a typewriter:

```

10 IF INKEY$ <>"" THEN GO TO 10
20 IF INKEY$ = " " THEN GO TO 20
30 PRINT INKEY$,
40 GO TO 10

```

Here line 10 waits for you to lift your finger off the keyboard and line 20 waits for you to press a new key.

Remember that unlike INPUT INKEY\$ doesn't wait for you. So you don't type ENTER but in the other hand you don't type anything at all then you've missed your chance.

INKEY\$ is very useful for a control loop where you can set objects on the screen to move according to what key you're pressing (for example the arrow keys). As you will also see in Chapter 20, one more option to you is to use the NEXT # TO keyword that works in a very similar manner. Finally it's also possible to query the keyboard hardware directly as well as the optional mouse as you will see in Chapter 22.

Animation: a quick primer

Animation is defined as any process with which static objects or pictures are manipulated to appear as moving. The word itself comes from the Latin *anima* which means *life*, in as sense it's to convey the appearance of life and movement to otherwise static constructs.

In computers, this is achievable using the rapid succession of images faster than the eye can perceive. In the ZX Spectrum Next specifically, there are basically five methods of animation: one using mass storage frame playback, the other using memory based frame playback, the third using sprites, the fourth using scrolling and the fifth is to use a combination of all the above. Let's examine them in turn.

Mass Storage Frame Playback

This technique deals with reserving part of complete frames on screens stored on your SD card or RAMdisk in the screen memory in rapid succession at the maximum possible speed. Consider this example using the RAMdisk:

```

10 INK 5 PAPER 0 BORDER 0 CLS
20 FOR f=1 TO 10
30 CIRCLE f*20,150,f
40 SAVE "a ba (" + STR$ (f) CODE
    16384 2048

```



```

50 CLS
60 NEXT f
70 FOR f=1 TO 10
80 LOAD 'm ball'+ STR$ (f) CODE
90 NEXT f
100 BEEP 0 01, 0.01
110 FOR f=9 TO 2 STEP -1
120 LOAD 'm ball'+ STR$ (f) CODE
130 NEXT f
140 BEEP 0 01, 0.01
150 GO TO 70

```

The example above works only on Layer 0 and leverages the RAMdisk without getting into BANK management territory. It can do that because the frames we're saving are very small. If you remember from Chapters 14 through 16 how the Layer 0 memory is organised in thirds, you'll soon figure out that although small it's not necessarily the faster way of doing things.

The RAMdisk is good to replay things but our SD card is also quite good. Let's try the following example with something more complicated based on a program contributed by mathematician Jwe Geiken from the NextBASIC forum.

```

1 REM Based on Rotating Ellipses by
  Jwe Geiken © 2019
10 RUN AT 3
20 LAYER 2 1 PAPER 0 CLS
30 X,Y=128 88
40 A,B=20 0
50 ITER,CURITER=20,0
60 FOR Q=0 TO 2* PI STEP PI/ITER
70 INK 246 A,B=30,16 P=0 PROC
  ellipse (X,Y,A,B P
80 INK 155 A,B,P=19 10,2* PI 0
  PROC ellipse (X,Y,A,B,P)
90 IF CURITER <=ITER THEN SAVE
  ANIM +STR$ (CURITER)+"_SL2"
  LAYER CURITER +=1
110 PRINT AT 23,0, 'Frame ',
  CURITER 1, ' saved', CLS IF
  CURITER / ITER THEN GO TO 220
120 NEXT Q GO TO 220
130 DEFPROC ellipse (X,Y,A,B,P)
140 LOCAL c,d,i,j,k,s
150 c,d=COS P,SIN P
160 FOR k= 0 TO 2.05* PI STEP PI /20
170 i,j=A* COS k,B* SIN k
180 IF k=0 THEN PLOT X+i*c-j*d
  y+i*d+j*c GO TO 200
190 DRAW X+i+c j*d PEEK 23426
  y+i*d+j*c PEEK 23430
200 NEXT k
210 ENDPROC
220 FOR %I = 0 TO 5

```



```

230 FOR J=0 TO ITER
240 LOAD 'ANIM'+STR$(J)+".SL2"
    LAYER
250 NEXT J
260 NEXT %I
260 LAYER 2 0 LAYER 0

```

The program generates ellipses that rotate counter to one another and after drawing each frame, saves the entire screen on the SD card. Once it's done generating (when CURITER reaches 100) it uses **LOAD LAYER** (which we will look at in depth in Chapter 19) to load and display the Layer 2 screens the previous part generated. Unlike the previous example using Layer 0 which only moved 2K at a time, this loads and displays 48K at a time.

Compared to the previous example using the *RAMdisk*, this appears much smoother and the reason is simple: there are many more frames generated by the program than what the previous one did. The question is can it be made smoother and if at all possible, faster?

Memory Based Frame Playback

It's time to delegate frame playback to RAM. Replace line 90 with this longer version:

```

90 IF CURITER <= ITER THEN SAVE
    'ANIM'+STR$(CURITER)+".SL2"
    LAYER BANK 9 COPY TO
111 (CURITER*3) BANK 10 COPY TO
110-(CURITER*3) BANK 11 COPY TO
109 (CURITER*3): CURITER +=1

```

and then add the following lines at the end:

```

270 PRINT AT 22,0,"Done Loading
    from SD. Press any key to
    load from memory."
280 PAUSE 0
290 FOR %I=0 TO 5
300 FOR %J=0 TO %INT(ITER)
310 BANK %111-(3*%J) COPY TO %9
320 BANK %110-(3*%J) COPY TO %10
330 BANK %109-(3*%J) COPY TO %11
340 NEXT %J
350 NEXT %I
360 LAYER 2 0 LAYER 0

```

Run the program again and now compare the playback using the SD card, with the playback of all the screens using the memory.

You can see that the playback is even smoother AND faster than the SD card and the reason is simple and that is because memory is a much faster medium than your SD card. Now there are several things of note here. First of all, this is not very efficient code: memory wise, Layer 2 uses 3 banks of 16k each making an entire screen 48K long. For the 20 iterations we made, that's $20 * 3 * 16K = 960K$ making this program unlikely to work on a non-expanded X68000 Spectrum Next. Secondly, not the entire screen is moving. Only a small window does and that makes saving the remainder of each screen wasteful in mem-

*You might be asking: "How powerful is a bank anyway?" It will work. Once we already know that banks exist, it's just using stuff by the system and CPU. It gives us a figure of 480K which is a minimum not available on an unexpanded Next.

only and speed. If we modify the program to draw the ellipses in one third of the screen (vertically speaking), we can only use 16k at a time making the program playback much faster. This is essentially the same thing the first program did using the *RAMDisk*. The line however appears jerky because there are not enough frames of animation to make our eyes be fooled by the illusion of smooth movement.

We can do that using **BANK LAYER** which is used to quickly copy data from a memory bank to the screen or vice versa. The syntax is as follows:

BANK n LAYER x,y w,h offset TO (raster op) offset x,y,w,h

which can copy any rectangular window of the current layer defined by x,y,w and h into a memory bank and back. **BANK LAYER** also supports effects defined by raster op which can further enhance the display of the "window" you're copying making an invert or trans image even more interesting. More information regarding **BANK LAYER** can be found in Chapter 23 *The Memory*.

Animation with the Sprite System

The third way of animating things in *NextBASIC* is via the use of the *Sprite System*. Sprites are visual objects of a rectangular shape that can be placed anywhere on the screen and animated by moving them about, but also performing animation within the object by simply replacing the object's bitmap (the image or pattern) displays. There are two kinds of sprites on the ZX Spectrum: *Next* 8-bit and 4-bit. The first can display 256 colours at once while the second 16.

There is a maximum of 128 sprites and 64 sprite patterns in 8-bit mode and 128 in 4-bit mode. *NextBASIC* only supports the 8-bit mode sprites so we'll only discuss these. For more information regarding the use of 4-bit sprites refer to Chapter 22 and *sprite at spectrad.com*. Information on 4-bit sprites is also included in the second volume of this manual.

Sprites are 16 x 16 pixels in size and can be mirrored and rotated. They can also be anchored together to make a bigger sprite.

The *Sprite System* has its own RAM located inside the FPGA that's accessible only from pureB which not accessible from the outside via standard **PEEK** and **POKE**. One can only write to it via **REG** commands and the special sprite ports. See Chapter 22 for details, so we need to keep a copy of our sprites in memory if we want to modify and send them to be displayed anew.

Creating Sprites

Sprites are created very similar to the way JDGs are created as we saw in Chapter 13.

There are three major differences however:

- JDGs are 1-bit only while sprites (for *NextBASIC*) are 8-bit
- JDGs are 8 x 8 while sprites are 16 x 16 pixels
- JDGs are manipulated within the main memory map while sprites need to be stored in a bank in order to be used

The similarities however are obvious. Sprites can be easily made with **DATA** statements which using one of the wider display modes—can even be seen visually via the numbers.

So where in a JDG you would 8 **DATA** statements of 8 bits each, in a sprite you would 16 **DATA** statements of 16 bytes each, the same essential thing but scaled up.

This is now demonstrated visually so let's try to implement the following sprite via DATA statements



Fig. 15 A sprite

The transparency (the large magenta-coloured area) is set to index 227 (as we've seen in Chapter 14, the *Global Transparency Colour*) which for the purposes of our example has been left the default. The rest displays a little spaceship in brown and gray while the cockpit is demonstrated in blue and white.

Let's start with the DATA statements. Some line numbers are omitted as we'll be adding them in the course of our animation example.

```

10 ; Sprite Romylos Dekos © 2019
30 RESTORE
40 BANK NEW a
50 FOR F=0 TO 255
60 READ r BANK a POKE f,n
70 NEXT f
80 SAVE "spaceship.spr" BANK a 0,255
210 REM Sprite Pattern 0
220 DATA 68, 68, 68, 68, 227, 227,
      227, 227, 227, 227, 227, 227, 68,
      68, 68, 68
230 DATA 68, 182, 219, 68, 227, 227,
      227, 68, 68, 227, 227, 227, 68,
      219, 182, 68
240 DATA 68, 68, 68, 68, 227, 227,
      227, 55, 55, 227, 227, 227, 68,
      68, 68, 68
250 DATA 182, 182, 68, 227, 227,
      227, 227, 55, 55, 227, 227, 227,
      227, 68, 182, 182
260 DATA 68, 68, 68, 227, 227, 227,
      68, 68, 68, 68, 227, 227, 227,
      68, 68, 68
270 DATA 240, 68, 68, 227, 227, 227,
      68, 255, 127, 68, 227, 227, 227,
      68, 68, 240
280 DATA 227, 68, 68, 0, 227, 227,
      68, 127, 127, 68, 227, 227, 0,
      68, 68, 227
290 DATA 227, 182, 219, 72, 0, 227,
      182, 0, 68, 68, 227, 0, 72, 219,
      182, 227

```



```

300 DATA 227, 182, 219, 72, 182, 0,
    0, 0, 68, 182, 227, 182, 72, 219,
    182, 227
310 DATA 227, 182, 219, 72, 182, 68,
    68, 0, 68, 68, 68, 182, 72, 219,
    182, 227
320 DATA 227, 240, 68, 72, 182, 68,
    68, 0, 68, 68, 68, 182, 72, 68,
    240, 227
330 DATA 227, 227, 227, 72, 182, 68,
    255, 182, 182, 255, 68, 182, 72,
    227, 227, 227
340 DATA 227, 227, 227, 227, 68, 68,
    255, 68, 68, 255, 68, 68, 227,
    227, 227, 227
350 DATA 227, 227, 227, 227, 227, 68,
    255, 182, 182, 255, 68, 227, 227,
    227, 227, 227
360 DATA 227, 227, 227, 227, 227,
    227, 236, 224, 236, 224, 227,
    227, 227, 227, 227, 227
370 DATA 227, 227, 227, 227, 227,
    227, 227, 252, 252, 227, 227,
    227, 227, 227, 227, 227

```

you use the 64 or 85 column modes (via the *Edit Options* menu, you'll be able to discern the pattern in a similar manner as you did for the IDs in Chapter 13. Value 227 is obviously the transparency as we discussed above.

Line 40 is a new command for us, which we will examine in length in Chapter 23, but what it does is to reserve the first free memory bank and assign its identification number to variable *a*. This way we don't need to remember (or hard code) an arbitrary number as that number could be in use if this is loaded on another machine.

Next, line 60 reads each value in succession and then writes, with **BANK POKE**, each value in a progressively increasing offset in bank *a*. Once the **READ** process is done, we **SAVE** the stored values in a file for later use. This particular version of **SAVE BANK** will be explained in length in chapters 19 and 23.

Putting Sprites on Screen

The sprite (or rather a *pattern* that can be assigned to a sprite) is now safely stored in bank *a*. So how do we display it?

For that we need a few commands: **SPRITE CLEAR**, **SPRITE BANK**, **SPRITE PRINT**, **SPRITE BORDER** and finally **SPRITE**.

Let's follow them one by one.

SPRITE CLEAR

clears all sprite assignments and starts fresh. It's a good idea to start any program dealing with sprites with that command so let's insert it into our program immediately with

```
20 SPRITE CLEAR
```

We now have at *TextBASIC* know that we have no sprites assigned with the previous command, but now we need to assign new ones. This is done with

SPRITE BANK *b*, *o*, *p*, *n*

which lets NextBASIC know in which bank *b* are the sprite patterns located. Optionally you can define a number *n* of sprite patterns beginning with pattern *p* located at bank offset *o*.

In the case above, we already know the bank and we do not need any more identification factors so let's tell NextBASIC where we put the sprites by adding

90 SPRITE BANK 0

All is now left to do is show our sprite. For this we need two commands. First we need to enable sprites with

SPRITE PRINT *n*

where *n* can be 0 or 1 enables sprites, 1 or disables (0) them. This is actually showing the sprites, but freshly initialised sprites contain no image (pattern), nor display information. We need to assign at least one pattern to one sprite "slot" and tell the Sprite System that the particular sprite "slot" is visible for that to happen.

In our example so far (that will soon change) we only have one pattern so that's not particularly difficult. We also need to place the sprite somewhere on the screen. And? possibly rotate it. If you go back to our sprite design, you'll see it's a spaceship facing upwards. We may need to make it turn to the left or right. All of the above (and one more thing) can be achieved with a single command:

SPRITE *s*, *x*, *y*, *p*, *f*, *rf*, *mx*, *my*

which in one go sets sprite number *s* to pattern number *p*, then update its position to location *x*, *y* with flags *f*, relative flags *rf*, x-scaling *mx* and y-scaling *my*.

If the sprite id *s* is negative, this sprite is a relative sprite, and its position is relative to the previous anchor sprite, defined with a positive sprite id. See later for more information on relative sprites. Flags is a bitmask (we've covered bitmasks before in Chapter 6 so that should be easy already) that sets the following:

Bit 0 is the *visibility* flag. 0 is for invisible and 1 is for visible.

Bit 1 is the *rotate* flag. 0 for standard, 1 for a 90° clockwise rotation.

Bit 2 is the *y-mirror* flag. 0 is for non-mirrored vertically while 1 is for mirrored.

Bit 3 is the *x-mirror* flag. Again it's 0 for non-mirrored horizontally while 1 is for mirrored.

while

Bits 4 through 7 define a 4-bit *palette offset* (or 0).

The Relative Flags *rf* is also a bitmask that sets the following:

The relative-flags parameter *rf* is also a bitmask.

Bit 0: type. 0=composite, 1=unified (only valid for anchor sprites).

Bit 1: pattern is relative to the anchor (only valid for relative sprites).

Bit 2: palette offset is relative to the anchor (only valid for relative sprites).

The scaling parameters *mx* and *my* are:

0= *x* (no scaling)

1= 2*x*

2= 4*x*

3= 8*x*

Any parameters in the **SPRITE** command can be omitted and its value will be left unchanged from the last time it was explicitly specified. It's a good idea again to use the **Bin** function to easily specify the flag parameters in a more convenient way.

We'll explain in a little bit the part about the *palette offset* but for now let's add a non-mirrored, non-rotated sprite 0 with the pattern 0 we defined, put it at approximately the centre

of our screen and make it visible. Let's add the appropriate commands now in our program:

```
100 SPRITE PRINT 1
110 SPRITE 0,152,119,0,1
```

to make sure that our sprite will stay on screen (as the NextBASIC editor will make it invisible temporarily when invoked) we should add one more line:

```
120 PAUSE 0
```

which will ensure the computer is waiting for our keypress before returning to NextBASIC. Now **RUN** the program.

Presto! Our spaceship is sitting idle, doing nothing in the middle of our screen. But wait a second? 152 and 119 don't look anywhere like the middle of the screen. We know our resolution in Layer 0 can be expressed in values between 0 and 255 for x and 0 and 191 for y, correct? Well wrong. It's time now to refer back to Chapter 14 and also examine Figure 1.0 one more time where we will see that the Sprite System has a resolution of 320 w x 256 h pixels.

This gives us 32 more pixels on every side than our standard resolution Layer 0 and Layer 2 screens. Now placement of the sprite begins with the upper left corner and a sprite is 16 x 16 so in order to be placed at the centre of the screen you divide the horizontal and vertical in half and then subtract a further 8 pixels to center the sprite. Normally the border hides the sprites so setting an x,y set of 0,0 would leave the sprite invisible. There is something we can do about that however and that's use

SPRITE BORDER n

which sets the sprites to print over the border if n is set to 1 or under if n is set to 0. Let's try it by adding the command and changing line 110 to show the sprite at that coordinate with

```
105 SPRITE BORDER 1
110 SPRITE 0,0,0,0,1
```

To execute with the latest changes do not **RUN** the program again as this will repeat the process and commit one more bank to the sprite **DATA** we introduced originally. Instead type **GO TO 100**. You may even want to test this without line 105 to see the difference. One more command related to the above is

SPRITE DIM x1,y1,x2,y2

which sets the clip window for sprites from x1,y1 to x2,y2. Any part of a sprite outside this window is not visible. Note that this has no effect if sprites over the border (**SPRITE BORDER 1**) is enabled.

Animating Sprites

This chapter however is called Time and Motion and with sprites so far we haven't seen motion at all. Well, let's change that as we spoke in the introduction a sprite can be animated by moving about the screen or by changing its bitmap to something different and most of the time both at the same time. In order however to animate the bitmap of a sprite a new pattern has to be defined. Let's do that by adding a few lines to our program and modifying some existing ones. First remove lines 140 and 150 then modify these

```
50 FOR F=0 TO 511
60 SAVE "spaceship.spr" BANK a,0,512
```

and then add these

```
106 FOR %a= 1 TO 50
130 %s=1 s
```



```

140 SPRITE 0,152,119,%a,1
145 NEXT %a
150 PALSE 0 STOP REM Exit here after
    pausing
155 REM Sprite Pattern 1
190 DATA 68, 68, 68, 68, 227, 227,
    227, 227, 227, 227, 227, 227, 68,
    68, 68, 68
400 DATA 68, 219, 182 68, 227 227,
    227, 68, 68, 227, 227, 227, 68,
    182, 219, 68
410 DATA 68, 68, 68, 68, 227, 227,
    227, 55, 55, 227, 227, 227, 68,
    68, 68 68
420 DATA 182, 182, 68, 227, 227, 227,
    227, 55 55, 227, 227, 227, 227,
    68, 182, 182
430 DATA 68, 68, 68, 227, 227, 227,
    68, 68 68, 68, 227, 227, 227,
    68, 68, 68
440 DATA 240, 68, 68 227, 227, 227,
    68, 255, 127, 68, 227, 227, 227,
    68, 68, 240
450 DATA 227, 68, 68 0, 227, 227,
    68, 127, 127, 68, 227, 227, 0,
    68, 68, 227
460 DATA 227, 182, 219, 72, 0, 227,
    182, 0, 68, 68, 227, 0, 72, 219,
    182, 227
470 DATA 227, 182, 219, 72, 182, 0,
    0, 0, 68, 182, 227, 182, 72 219,
    182, 227
480 DATA 227, 182, 219, 72, 182, 68,
    68, 0, 68, 68, 68 182, 72 219,
    182, 227
490 DATA 227, 240, 68, 72, 182, 68,
    68, 0, 68, 68, 68, 182, 72, 68,
    240, 227
500 DATA 227, 227, 227, 72, 182, 68,
    255, 182, 182, 255, 68, 182, 72,
    227, 227, 227
510 DATA 227, 227, 227, 227, 68, 68,
    255, 68, 68, 255, 68, 68, 227,
    227, 227, 227
520 DATA 227, 227, 227, 227, 227, 68,
    255, 182, 182, 255, 68, 227, 227,
    227, 227, 227
530 DATA 227, 227, 227, 227, 227,
    227, 236, 224, 236, 224, 227,
    227, 227, 227, 227, 227

```



```

540 DATA 227, 227, 227, 227, 227,
      227, 227, 224, 224, 227, 227,
      227, 227, 227, 227, 227

```

Now, unlike the previous encouragement, **RUN** the program again. This will reserve a new bank for sprites which isn't normally recommended but it is okay for the purposes of our example. What we have done now is to create two patterns that are similar but differ slightly in the cannons section and the engine section. Lines 136 to 150 will display sprite 0 50 successive times, however where things differ is at line 137 which flips a switch from pattern 0 to pattern 1 for sprite 0 displayed at line 140. If you cannot see the effect, very well, you can insert a

PAUSE 3

at the end of line 140 which should give you just about enough delay to see the sprite changing at the engine and cannon sections while at the same time demonstrating how important time control is in animation. We did cover the bitmap animation of the sprite itself, let's now see how we can make it move. First however let's try to rotate the sprite in place so we can also see the usage of the flags in action. Add the following lines

```

107 %p=0
108 REPEAT
109 IF %p=0 THEN %f=%000001
110 IF %p=1 THEN %f=%00011
111 IF %p=2 THEN %f=%00101
112 IF %p=3 THEN %f=%01011
141 REPEAT UNTIL %p=3

```

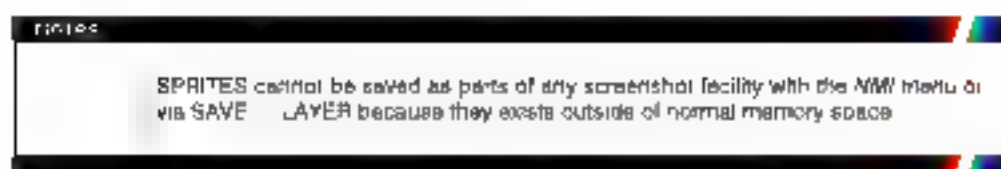
and make line 140

```

140 SPRITE 0,152,119 %s,%f PAUSE 3
      %p+=1

```

Now execute again with **GO TO 100** and you will see the sprite rotate in place



The process is quite simple, the last bit being the visibility flag

First the sprite is printed upright, then the rotation flag bit gets turned on to give it a right angle turn, then it gets turned off and the Y mirror flag bit gets turned on to make the sprite point downwards and finally the rotation flag bit together with the X mirror flag bit get turned on to rotate the sprite clockwise 90° and then mirrored horizontally to make the sprite pointing to the left. The process restarts from the sprite pointing upwards when the rotation variable %p gets reset to 0 and the whole thing repeats 50 times, all the while changing between patterns 0 and 1.

Moving Sprites on Screen

Time to move the sprite about the screen, we'll start easy and then introduce you to the real reason (that is obviously humorous, why maths exists). First remove all lines between 06 and 150 and replace with these

```

106 FOR %a = 0 TO 255

```



```

130 %s=%1 5
140 SPRITE 0,152,%255 a,%s,1
141 PAUSE 3
145 NEXT %a
150 GO TO 105 REM you'll need to
    stop this with BREAK

```

Execute with **GO TO 100** and you'll see our spaceship fire up its engines and cross the screen from top to bottom. Now for something much fancier as promised: move line 105 to 120 and add these lines:

```

105 PROC initSXsineMov()
550 STOP
570 DEFPROC initXSineMov()
580 FOR f=0 TO 319 %a[ f ]=INT ( f / 2 * INT
    (159 * SIN ( f / 159 * PI )) ) NEXT f
590 ENDPROC

```

Finally modify lines 140 and 150 as follows:

```

140 SPRITE 0,%159+a[a],%255-a %s,1
150 GO TO 120

```

Before executing again with **GO TO 100**, the spaceship now will move in a sinusoidal pattern from the bottom of the top of the screen before wrapping around and coming from the bottom. The way we did this was by precalculating an integer array (see Chapter 2) to hold all possible *x* values within our visible Sprite System coordinates. To avoid **Integer out of range** errors, we made sure the possible values of both the **SIN** function results and the *f* that positions the spaceship on the *x,y* axis stay within an acceptable range. To switch the initial direction of movement, instead of *a +* you can start with *a -* in line 140 as follows:

```

140 SPRITE 0,%159-a[a],%255 a,%s,1

```

Note that our integer array *%a* is using the brackets *f* variable instead of the parentheses variable *a* that's because we have more than a hundred 64 values. *f* also means also that integer arrays *%a*, *%b*, *%c*, *%d*, and *%e* have been used up by *%a*.

It's obvious by this example that very complex arithmetic contains can be created with relative ease using the Sprite System. Before we move on to scrolling, it's useful to also cover a few more subjects we did not address in the course of our example.

The first thing is the ability to use palettes with the Sprite System. These are indistinguishable from other palettes in the ZX Spectrum Next palette control system and they too are also governed by the **PALETTE DIM** keyword (set them up as 8 or 9 bits like the **LAYER PALETTE** equivalent). The Sprite System has its own keyword combinations: **SPRITE PALETTE** and **SPRITE PALETTE BANK**. The syntax is as follows:

SPRITE PALETTE *n*[,*v*]

where *n* is the palette number (0 for first and 1 for second) while the optional *v* are the colour index (0 to 255) and colour value (expressed in 8-bit RR GG BB format regardless of the **PALETTE DIM** setting).

SPRITE PALETTE *n* BANK *b*,*o*

will operate like it's **LAYER** counterpart, assigning palette *n* from bit set *o* in bank *b*. As with the **LAYER** version, palettes are 12 bytes long if 4-bit and 256 bytes long if 8-bit as set with **PALETTE DIM**.

One last thing of note is the palette offset flag we discussed earlier. This is there to allow for quick change of colour scheme on a sprite without changing its bitmap. If you recall the discussion about 4-bit sprites, this is similar but the sprites are actually 8-bit ones. They can still be defined in 8 bit index values however, these values' 4 top bits will get chopped off and replaced by the optional offset. Since calculating and/or anticipating and properly structuring your palettes for such a use can be a large hassle, it's good practice you want to use this feature to define your sprite values from 0 to 15 and set the offset to adjacent sets of 16 colours. This way in a potential future version of NextBASIC that supports native 4-bit sprites, you won't have to change pattern definitions at all.

Relative sprites

Sprites can be grouped together to form a *composite* or *unified* sprites. Each such grouping consists of a single *anchor* sprite which is the sprite with the lowest id in the grouping, followed by any number of *relative* sprites, with sprite ids following the anchor sprite in sequence.

When an anchor sprite moves or becomes invisible, all the associated relative sprites also move or become invisible. It is also possible for individual relative sprites to be made invisible or visible. The rule is that a relative sprite is only visible if its own visibility flag is set and the visibility flag of the associated anchor sprite is set.

To define a relative sprite, simply specify its sprite id as a negative number for example specifying

```
SPRITE -1
```

defines sprite 1 as relative to the preceding anchor sprite with id 0.

Any number of relative sprites can follow an anchor sprite.

The x and y coordinates specified in **SPRITE** commands for relative sprites are not actual coordinates, but signed integer offsets in the range -128 to +127 from the coordinates of the anchor sprite. This establishes how close the anchor and the relatives are, in other words they don't have to touch each other, only when we want to create something visibly bigger looking like one single thing on screen.

Additionally, if the pattern relative flag is set for a particular relative sprite, its pattern number is added to the pattern number from the anchor sprite, wrapping round if the sum exceeds 64.

Using this, it's easy to animate an entire composite/unified sprite simply by changing the pattern of the anchor sprite. This is extremely similar to our small animation example before.

In the same vein, if the palette relative flag is set for a particular relative sprite, its palette offset is added to the palette offset from the anchor sprite (wrapping round if the sum exceeds 16).

Composite vs Unified sprites

The type of a grouping of sprites is determined by the type flag of the anchor sprite, composite or unified. The distinction between them is simple.

For composite sprites, the remaining sprite parameters (rotation, x/y mirrors and x/y scaling) are independent for each relative sprite. This allows creation of a composite sprite where individual relative sprites can be rotated etc. for animation purposes, whereas for unified sprites, the rotation and x/y mirrors of the relative sprites are relative to that of the anchor sprite.

Therefore, when the rotation or x/y mirrors of the anchor are changed, all the relative sprites rotate or reflect about the anchor. The same goes for the x and y scaling of the indi-

visual relative sprites in the case of unified sprites, it is completely ignored, thus allowing the entire grouping to be scaled (as by changing the scaling of the anchor).

Batching

The standard **SPRITE** command normally has immediate effects to what is displayed on the screen. However, it is also possible to place NextBASIC into *batching mode*.

In this mode, **SPRITE** command has no immediate effect, but the changes specified are remembered. When all the required changes have been made using multiple **SPRITE** commands, they can all be applied to the screen at once, giving a more synchronised look to your game and fewer screen tears.

To control batching, the following commands are available in the order one would use them:

SPRITE STOP

which enables batching mode and turns off the immediate sprite screen updates.

SPRITE MOVE

This command sends all outstanding sprite changes to the hardware immediately (hence displays all the animation effects and movement that was pending while in batching mode).

SPRITE MOVE INT

The same as above but waiting for the 50 Hz / 60 Hz interrupt to occur. This is useful in order to synchronise the sprite movement to the frame rate, making for a smoother animation and finally:

SPRITE MOVE INT y

which works in the same way as **SPRITE MOVE INT** except that changes are not sent to the hardware until after the TV scanline corresponding to sprite coordinate y. This avoids flicker by making sure that sprite changes do not happen on screen while the display is midway through displaying the current sprite(s). Finally:

SPRITE RUN

disables batching mode and turns the immediate screen updates back on. This is also done by the **SPRITE CLEAR** command we saw earlier but without the destructive effects.

Automatic sprite movement

As we saw above, moving a sprite can be laborious. In order to reduce the amount of work a NextBASIC program needs to do to animate and move sprites, commands are provided to allow some or all of this work to be done automatically whenever a **SPRITE MOVE** command is issued. Any sprite can have automatic movement or animation applied to it, and the standard **SPRITE** command can still be used to perform any other changes when they are needed.

The main command used to set up automatic sprite movement is the **SPRITE CONTINUE** command:

SPRITE CONTINUE *x1* [TO *x2*] [STEP *xs*] **RUN STOP** *y1* [TO *y2*] [STEP *ys*] **RUN STOP** [*p1*] [TO *p2*] [*f1*] [*f2*] [*d*]

Although looking somewhat daunting at first, **SPRITE CONTINUE** is quite easy to master regardless of its numerous options. As apparent by the brackets, each parameter (or sub-clause of a parameter) is optional. If not specified, the previous value will be retained.

Movement in the x-direction is specified with

3 The scaling also applies to the relative x/y coordinate offsets.

x1: minimum value for *x*-coordinate
x2: maximum value for *x*-coordinate (or assumed to be equal to *x1* if not provided)
xs: signed step in pixels (between -127 +127) for every horizontal move
 and
RUN: indicates movement in the *x*-direction is initially on or
STOP: indicates movement in the *x*-direction is initially off

The parameters are equivalent for the *y*-direction and specified with

y1: minimum value for *y*-coordinate
y2: maximum value for *y*-coordinate (or assumed to be equal to *y1* if not provided)
ys: signed step in pixels (between -127 +127) for every move
RUN: indicates movement in the *y*-direction is initially on
STOP: indicates movement in the *y*-direction is initially off

Pattern animation is specified with

p1: minimum value for sprite pattern
p2: maximum value for sprite pattern (or *p1* if not specified)

while movement rates are controlled with

r: rate at which sprite moves/animates (0-255) where
 0 = on every **SPRITE MOVE** command
 1 = skip 1 **SPRITE MOVE** command after moving
 2 = skip 2 **SPRITE MOVE** commands after moving
 etc

d: delay before initial movement (0-255), where
 0 = move on the first **SPRITE MOVE** command
 1 = skip 1 **SPRITE MOVE** command before the first move
 2 = skip 2 **SPRITE MOVE** commands before the first move
 etc

The flags parameter *f* is an 8-bit mask which as seen before is best specified with **Bin** or **@**:

bits 1-3 define the behaviour when the *x,y* limits are reached
 00 = reflect this direction
 01 = stop this direction, start other direction
 10 = stop this direction
 11 = stop completely and make sprite invisible
 bit 2 flips the *Y-mirror* flag when *y* limits are reached
 bit 3 flips the *X-mirror* flag when *x* limits are reached
 bit 4 controls the behaviour of pattern change
 0 = cycle upwards, wrapping back to lower limit
 1 = bounce between lower and upper limits
 bit 5 if set, sprite is disabled when its pattern reaches limits
 bit 6 if set, updates the pattern even when sprite is stationary and initially
 bit 7 if set, *X-mirror/mirror/rotation* flag are set according to the direction of travel
 (this bit overrides bits 2 & 3)

The initial position, pattern and other details of the sprite are determined by the last standard **SPRITE** command. If these values are outside the maximum/minimum ranges, then (depending upon the specified *step* and **RUN/STOP** status) they will gradually change until they fall within the max/min range.

Automatic movement is specified for an anchor sprite, then the entire composite or unified sprite will be automatically moved.

Automatic movement can also be specified for individual relative sprites if this is desired. (Since the parameter *s* is always positive for the **SPRITE CONTINUE** command, but the sprite remains relative if specified as such in the last standard **SPRITE** command).

This would be done to animate the relative sprites separately, so that one relative sprite might animate whilst the others remain the same (for example). However, it can also be used to automatically move a relative sprite around within a composite/group sprite. The main restriction here is that the mover's offsets are unsigned, so this works best for relative sprites with positive offsets: add 256 to negative offsets when specifying them in a movement range.

Automatic movement can be temporarily suspended for particular (or all) sprites if so desired by using

SPRITE PAUSE *s1* [TO *s2*]

which turns off automatic movement for a single sprite *s1* or a range *s1* - *s2* of sprites. Said suspension is lifted by using

SPRITE CONTINUE *s1* [TO *s2*]

which restarts automatic movement for a single or a range of sprites as above.

Sprite functions

In order to return details about sprites, and to make collision detection checks, several functions are provided*. These are

SPRITE *s*

which shows if sprite *s* is visible. Returns 1 (true) if sprite *s* is visible or 0 (false) if not.

SPRITE CONTINUE *s*

Returns a bitmask describing the automatic movement enabled for sprite *s*:

bit 0: set if automatic movement is enabled

bit 1: set if currently moving in the Y axis

bit 2: set if currently moving in the X axis

Note that if bit 0 is set but neither bits 1 or 2 are set, that means that only the pattern is being animated.

SPRITE AT(*s,c*)

Returns a coordinate or other movement-related value for the sprite

SPRITE AT(*s,0*) returns *x coordinate*

SPRITE AT(*s,1*) returns *y coordinate*

SPRITE AT(*s,2*) returns *pattern number*

SPRITE AT(*s,3*) returns *x step*

SPRITE AT(*s,4*) returns *y step*

SPRITE AT(*s,5*) returns *delay before the sprite next moves*†

One of the most labour-intensive game programming tasks is trying to figure out when two sprites have collided. NextBASIC provides that information with

SPRITE OVER(*s1, s2* [TO *s3*] [,*overlapX*] [,*overlapY*])

which performs a bounding-box collision detection between sprite *s1* and a single other sprite *s2* or a range of sprites *s2* - *s3*.

Two optional, acceptable overlaps (in pixels) can be provided in *overlapX* and *overlapY*. If *overlapX* is not present, 0 (no overlap) is used; if *overlapY* is not present, then the value of *overlapX* is used. Overlaps should be 0-7 for an unscaled sprite *s1*, or 0-15 for a 2x scaled sprite etc. Overlaps allow for some flexibility before declaring a collision especially since sprite patterns do not always extend to the boundaries of the sprite bounding box (16 x 16).

* All these functions are available in the standard expression evaluator and the integer expression evaluator.

† A returned value of 0 means the sprite will move on the next **SPRITE MOVE** command.

The function returns 0 (false) if there is no collision or the number of the colliding sprite (s2 - s3) if there was a collision.

NOTE If the colliding sprite's id is 0 then 128 is returned.

Any relative sprites following s2 or s3 will also be checked, until the next anchor sprite that is not in the specified range.

Scrolling

The easiest method of animation is by using the in-built hardware scrolling capabilities of the ZX Spectrum Next. As you will find out in Chapter 22, all layers can be scrolled either in full or within a clipping window (see Chapter 16 Graphics). NextBASIC provides access to hardware scrolling via the **LAYER AT** command. Its syntax is as follows:

LAYER AT x,y

which moves the current layer to the offset defined by the coordinates x and y. According to which side we're moving to, the existing graphics on that side get wrapped around the opposite side. Let's demonstrate using one of the images we generated earlier while doing frame-based animation:

```
10 LAYER 2,1 CLS
20 LOAD 'ANIM0.SL2' LAYER PAUSE
   0 REM Hasta la Vista Key
30 FOR %X=0 to 255
40 LAYER AT %X,%0
50 NEXT %X
60 LAYER AT 0,0. LAYER 2,0 LAYER 0
```

Once you **RUN** the above, you'll see an image racing towards the left side of the screen so fast it may even be unusable. If anything, it'll have a simple effect. Running it at 3.5MHz you will see a very smooth movement which shows how efficient hardware scrolling is on the ZX Spectrum Next.

If you want to reverse the effect and make the screen move towards the right you will need to change line 40 to:

```
40 LAYER AT %255-%X,%0
```

If we borrow a bit from the sprite example, we can even introduce a **SIN** function to make the screen appear like it's bouncing from left to right and up to down, and vice versa.

By itself, the **LAYER AT** keyword doesn't do much other than roll a screen around, with the combination however of layer clipping windows and background updating of the shadow screens (See Chapters 22 and 23 as well as Chapter 16) you can produce a scrolling effect of very large landscapes. If you combine this with specially crafted screens that can repeat themselves at infinitum then you have the basics for every side scrolling game ever made!

The Copper

While not strictly an animation aid, the Copper is a hardware module of the ZX Spectrum Next that can definitely be used for among other things, animation. The Copper runs in parallel and independently from the main Z80 processor and is dedicated to writing Next Registers (NextREGs) at specific points on the display. The name derives from cop-processors and was first seen in the Amiga computer which had a similar function. The Copper essentially maintains a list of instructions that consists of only two commands: **WAIT** and **MOVE**. This simple control allows updating of Next registers at regular times, synchronised to points when the display is updated on the screen. The Copper system can therefore be used to send audio samples to the ZX Spectrum Next's digital audio hardware.

make 'as' colour changes to get sky effects, change layer priorities, enable or disable screen modes etc. all that from a simple list of commands.

On older Spectrum models, you would have needed some very clever use of the interrupt system to do these sort of tricks with some being completely impossible or just too slow to be of any practical use. Even with the ZX Spectrum Next's ability to generate interrupts on each raster line, setting that up (especially in *NextBASIC*) and then trying to get the timing right for nice clean effects is very complicated, or impossible, and yet simple to accomplish by using the Copper.

We'll jump ahead a bit and introduce a special command **REG** (which will be covered in full in Chapter 22). For now, take **REG n,v** to be the same as **OUT 9275, n: OUT 9531, v**. Let's see our example:

```

10 BORDER 0 PAPER 0 INK 7 CLS
20 REG 98 0 REM make sure Copper is
   stopped
30 REG 97,0
40 REM Select the Copper data
   register
50 FOR x=0 TO 5 REM Increase this
   if you add more data lines.
60 READ m,v
70 REG 96 m REG 96 v REM write the
   Copper list from DATA statements
80 NEXT x
90 REG 97,0 REM low part of address
100 REG 98,%011000000 REM high part
   of address and start Copper
   repeat on 0Blank
1000 DATA 128+(45*2),0
   REM WAIT for line zero horizontal
   45
1010 DATA 64,16,65,BIN 11100000
   REM WRITE Palette Index 16 (Paper
   and Border), then WRITE RED
1020 DATA 126+(45*2),100
   REM WAIT for line 100 horizontal
   45
1030 DATA 64,16,65,BIN 00000000
   REM WRITE Palette Index 16 and
   WRITE contents back to BLACK.
1040 DATA 126+1,128
1050 REM Last line waits for a bit of
   the screen that does not exist
   1*256+128 = 386 (STOP)

```

You can try changing the **BIN** statements in lines 1010 and 1030 to use different colours (this is the 8 bit Palette value so **RRRRGGGBB**).

Now remember this list is still running in the background but it is changing CLS's palette (0 paper colour). *NextZXOS* uses palette 1 so you can't see it when running *NextBASIC*. Just type **CLS** and you will see that it comes back until you press a key.

WAIT commands (where the top bit is 1 i.e. bytes > 128) will pause processing until a certain point on the display (to a fixed resolution).

MOV_i commands (where the _i bit is 0) the bytes < 128 will take a given value and put it in the numbered register.

You can have up to 1024 commands which can repeat or stop at any point by WAITing for a non-existent line (ie 11) which works at both 50 and 60 Hz. So there's loads of room for creativity and invention.

Only the lower 128 Next registers can be written but this is not an issue as the registers above 127 are mainly used for the accelerator and the Expansion Bus.

Register 96 (60h) is the data port to write the instructions. They are two bytes long so you need to write them in pairs with the most significant byte first (not the usual Z80 way but needed for the way the system works).

Register 97 and 98 (61h and 62h) are the controls. The first is the low 8 binary bits of the address to WRITE the instructions, the second contains the bits to control the mode and the top bits of the instruction address. If you change to mode 0 (b) (from another mode like 00: PAUSE/STOP) this also tells us where the Copper begins to READ its instructions from back to instruction 0. In all other cases it will carry on from where it left off last time.

The Copper sees the screen starting from the top left pixel of the display area of the screen. This is 10. After 32 horizontal values (every 8 pixels) you have the right border, then you have a gap (count of 2) which is where, on an old TV, the spot would be flying back over to the left. Then you have the right hand border of the text horizontal line.

Note: This zero point is also where the screen "dot" will be when the first Raster Line Interrupt occurs. Do not confuse this with normal interrupts on the system which occur in the top left of the whole screen as it is displayed on a monitor or TV. That is actually somewhere in the middle of the bottom right of the Copper view of the screen shown in the diagram below. Exactly at raster line 224 at 60Hz or 248 at 50Hz.

Finally when it gets to the bottom of the screen it has the border and then a blank period (8 lines) while the old spot was running back to the top of the screen. Then you have a number of lines in the top of the screen area to play with (56 at 50Hz or 32 at 60Hz). To see this change line 000 for DATA 128 + (45*2) 200 and line 1020 for DATA 128 + 45*2 + 1 45. Remember $1*256+45 = 301$.

This diagram will hopefully help to visualise that.

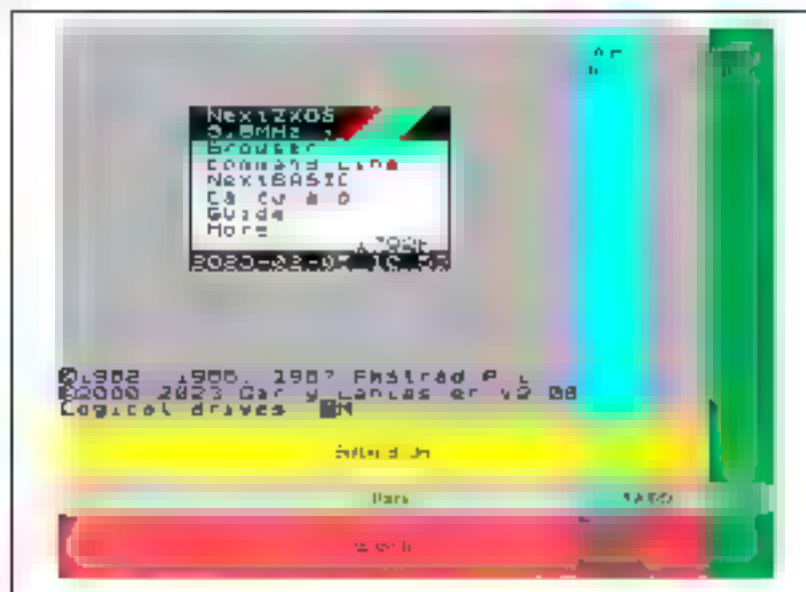


Fig. 18 Copper operation

your MOVE 0,0 (i.e. Write something to a Read Only Next Register like Register 0), then the Copper does nothing for a short duration: a NOP in Z80 terms, so you can wait for a more accurate moment to overcome the fact you only have 55 horizontal positions to wait for: i.e. every 8 pixels on the screen.

You can write to the Copper as it is running because it keeps a separate track of its READ instruction address to the address you are using to WRITE.

WARNINGS

Be careful as the NextZXOS Screensaver uses whatever palette is in place so if you have any border effects running they will still be visible and could cause a CRT screen to experience a burn-in effect. This is worth bearing in mind if you are writing software not to leave static images around too long.

If you try to write to a Next Register at the same time as the Copper then this might cause a conflict: don't worry, the Copper will win and the display will be OK but your program command may fail.

So some care is needed to manage the two systems. Turning off the Copper while you make Next Register addressing changes in NextBASIC is a good idea. That includes things like the PALETTE command for example. If you are using machine code you will need to use some form of flag and remember what the Copper might be doing at a specific time.

In the above program for example, it is possible the Copper STOP in the first two lines will fail if you run it a second or third time to change the colour and will not reset the write address, so you will write after the list already there and your new one will never be reached. You could get around that by repeating the first two lines as it is unlikely to fail twice so shortly after the last attempt and has no effect if it does run twice.

Exercises

- 1 Write a procedure to write a STOP command twice in a row so that you can make sure the Copper is stopped when you need to in your programs.
- 2 Draw a Spectrum Flash on the right hand side border by changing the palette colour five times. Make sure the last time is back to your real paper/border colour. Hint you can use one or more WRITE 0,0 as a very short delay.
- 3 Write a program that controls two spaceships using the sprite defined: one going horizontally while the other vertically on the screen.
- 4 Enhance the above program with a memory based Layer 2 animation running in the background.

Chapter 18 Sound and Music

Unlike its predecessors, your ZX Spectrum Next doesn't fare poorly in the audio capabilities department. From simple beeps and clicks to complex compositions using its in-built 3 Programmable Sound Generators (PSGs) and full-fledged digital audio output, sound can accompany almost every program you write or software you will load. Sound is output in stereo from both the digital video port and an analogue 3.5mm jack output present on the back of the machine. Additionally, there is the possibility of an on-board piezo speaker sold separately.

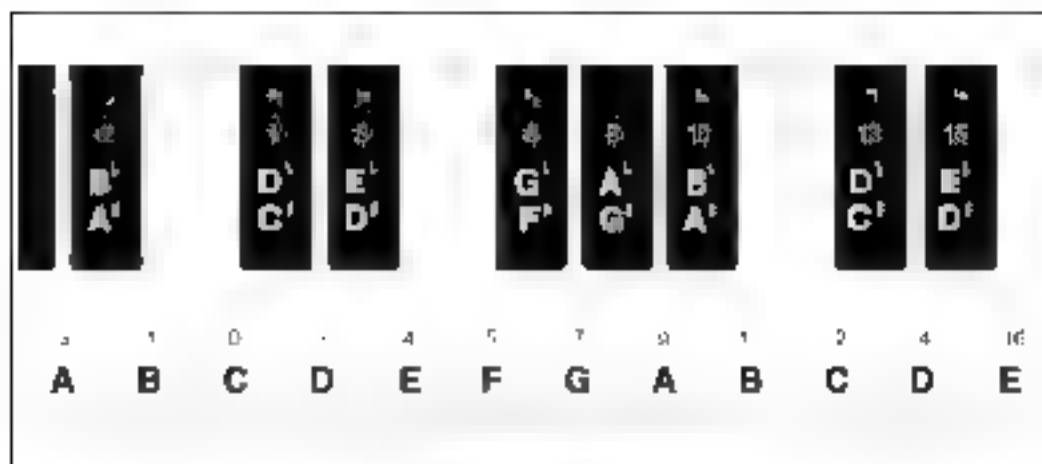
Basic sounds with the BEEP command

The easiest way to create sounds (and the only method that works on all ZX BASIC versions including NextBASIC) is by using the **BEEP** statement.

BEEP duration, pitch

where, as usual, *duration* and *pitch* represent any numerical expressions. The *duration* is given in seconds, and the *pitch* is given in semitones above middle C. For notes below middle C we use negative numbers.

Here is a diagram to show the pitch values of all the notes in one octave on the piano.



mine, since tone generation via the CPU is an exclusive task, you cannot do anything else on or off screen while the sound is playing, so in order to perform other functions while sound is generated by using the CPU, you will also have to program in Machine Code, or – assuming you have the Accelerated version of a Pi Zex – insalled, use the audio playback facilities described in the last section of this chapter (the latter working independently of whatever the ZX Spectrum Next is doing).

Try programming tunes in for yourself – start off with fairly simple ones like *Three Blind Mice*. If you have neither piano nor written music, get a very simple instrument, like a tin whistle or a recorder, and work the tunes out on that. You could make a chart showing the pitch value for each note that you can play on this instrument.

Type

```
FOR n=0 TO 1000 BEEP .5,n
NEXT n
```

This will play notes as high as it can, and then stop with error report **B** integer out of range. You can print out *n* to find out how high it did actually get.

Try the same thing, but going down into the low notes. The very lowest notes will just sound like clicks, and the higher notes are also made of clicks in the same way, but faster, so that the human ear cannot distinguish them.

Only the middle range of notes are really any good for music. The low notes sound too much like clicks, and the high notes are thin and tend to warble a bit.

Type in this program line

```
10 BEEP .5,0 BEEP .5,2 BEEP .5,4
   BEEP .5,5 BEEP .5,7 BEEP .5,9
   BEEP .5,11 BEEP .5,12 STOP
```

This plays the scale of C major, which uses all the white notes in the piano from middle C to the next C up. The way this scale is tuned is exactly the same as on a piano, the so-called even-tempered tuning, because the pitch interval of a semitone is the same all the way up the scale. A violinist, however, would play the scale very slightly differently, adjusting all the notes to make them sound more pleasing to the ear. He can do this just by moving his fingers very slightly up or down the string in a way that a pianist can't.

The *natural* scale, which is what a violinist would play, comes out like this

```
20 BEEP .5,0 BEEP .5,2.039 BEEP .5
   3.86 BEEP .5,4.98 BEEP .5,7.02
   BEEP .5,8.64 BEEP .5,10.86
   BEEP .5,12 STOP
```

You may or may not be able to detect any difference between these two – some people can. The first noticeable difference is that the third note is slightly flatter in the *natural* tempered scale. If you are a real perfectionist, you might like to program your tunes to use this *natural* scale instead of the even-tempered one. The disadvantage is that although it works perfectly in the key of C, in other keys it works less well – they all have their own natural scales – and in some keys it works very badly indeed. The even-tempered scale is only slightly off, and works equally well in all keys.

This is less of a problem on the computer, of course, because you can use the trick of adding on a variable key.

Some music – notably Indian music – uses intervals of pitch smaller than a semitone. You can program these into the **BEEP** statement without any trouble. For instance the quartertone above middle C has a pitch value of .5.

You can make the keyboard beep instead of clicking by


```
POKE 23609,255
```

The second number in this determines the length of the beep. Try various values between 0 and 255. When it is 0, the beep is so short that it sounds like a soft click.

Enhanced Sound and Music with PLAY

When using *NextBASIC*, you have two different ways to make music and sound effects. You can still use the **BEEP** command, as discussed above, but you also have access to the **PLAY** command which allows you to make much more sophisticated music, with up to nine notes playing at once. It also gives you more control over the sound of each individual note than is possible using **BEEP**.

Making music and sound effects with **PLAY** is simple. You just type in the series of notes that make up a tune, then ask the ZX Spectrum Next to **PLAY** them. You can also include instructions that tell your machine what sort of tone you want for the sound. Please note that case is important when typing in the string expressions in the examples i.e. **ga** should not be typed as **Ga**, **gA** or **GA**.

To hear some of the wide range of sounds that you can make, type in one of the two programs below. **RUN** it, then try the other example. Don't worry if the program lines look complicated, they are explained in detail later.

Music

```
10 b$='04 CDEC) (5EF7G) (3GAGF5EC)
   5EB7E9EBE
20 PLAY "T18006(CDEC) (5EF7G) (3GAGF5EC)
   5CG7C9CGC',b$, '03(7CG,7CG)(7CG)
   5GD7G9G9G"
```

Sound Effects

```
10 a$='M8UX350W507(((C)))"; PLAY a$
   PAUSE 25
20 PLAY "M56Jx5000L103((C))"; PAUSE 25
30 a$='M58W201N8C' PLAY a$; PAUSE
   25
```

Using the PLAY command

In the examples above, you will see that each time the **PLAY** command appears, it is followed by up to three different parameters in the form of either *string variables*, *string literals* or a combination of both in a statement like

```
PLAY "P1C2,P1C3,P2C1,P2C2,P2C3,X" P3C2, "
```

where **PxCy** are strings that refer to the PSG 'P' number (x: 1 to 3) and channel (C) number (y: 1 to 3). The order of these is specific and each **PLAY** command must have the full complement if you require all the channels to reproduce a sound. You cannot issue two or more **PLAY** commands to control individual PSGs, as each **PLAY** statement sends a batch of instructions to the audio hardware. If you wish one or more channels to be silent, you should replace them with the empty string "". As we will examine below, the strings contain all the information to tell your ZX Spectrum Next which sounds to make.

As we discussed, **PLAY** controls nine separate sound channels over the 3 available PSGs, each called **A**, **B** and **C**.

In the *Music* example given above, "T18006(CDEC)(5EF7G)(3GAGF5EC)5CG7C9CGC" tells channel **A** of PSG1 to play the melody line, b\$ tells channel **B** of PSG1 to play a harmony, and "03(7CG,7CG,7CG)5GD7G9G9G" tells channel **C** of PSG1 to play a bass part. In the *Sound Effects* example, only one noise is used at a time, although up to nine

can be – so each one is in channel **A** of PSG† and the command is simply **PLAY a\$** (or as seen in line 20) **PLAY "M56UX6000W103,(C),"**

† that any of the channels can produce either a musical tone or noise – in even mixing at all so you can mix sound effects in with your music – see Channel selection later on.

Constructing strings

Composing music and sound effects in *NexBASIC* is just a matter of creating strings containing the information you want. Try this – very simple – example which plays just one note – an **A**.

```
a$="a      PLAY a$
```

Any music program using **PLAY** will generally use string variables rather than literals to tell it what to play – as you can see by looking at the earlier examples. The more complex (or longer) the piece and the more complicated sound, the more complex the strings become – as obvious from the increased complexity of the examples above.

Any musical sound has a *pitch* and *duration* – it also has a *volume* and *timbre*. The strings in the earlier examples contain information about all of these. The summary below lists each possible command, and they are explained in detail opposite.

PLAY command summary

This is a brief list of the commands which can be contained in a **PLAY** string. Note that all letters except note names must always be in capitals.

| String entry | Function |
|------------------------|---|
| A or C-B | Gives pitch of note within current octave range |
| \$ | Flattens note (lowers it) |
| # | Sharpens note (raises it) |
| Ox | Sets octave range to x (-1 to 8) |
| D or D2 | Sets duration of note |
| A | Terminates a list |
| N | Separates two sublists |
| Vx | Sets volume to x (0-63) |
| Wx | Sets volume effect to x (0-7) |
| L | Turns on volume effect to the current channel |
| Xx | Sets duration of volume effect to x (0-65535) |
| Tx | Sets tempo to x (EQ-2400 bpm) |
| (| Enclose repeated phrase |
|) | Ends a repetition |
| H | Halts a PLAY command |
| Mx | Selects channel and sets type to x (-63) |
| Vx | Uses an MIDI channel as x (0) |
| Zx | Sends x as a MIDI pair |
| L | Restricts output from current PSG to Left Speaker Only |
| R | Restricts output from current PSG to Right Speaker Only |
| S | Resets stereo mode to current PSG |

Table 10 PLAY commands

Setting the pitch

As you saw above, you set the pitch of any note by giving its musical name – eg **C E G**. Sharp notes are prefixed by **#** (eg **#C**) and flat notes by **\$**. A two octave range in the key of **C** – which use the letters **c** to **b** for the notes in the lower octave and **C** to **B** in capitals for the

higher one are available at any moment. Any number of notes within these two octaves can be played one after another. For example:

```
10 a$='cfedafgCFEDAFGCC'
20 PLAY a$
```

If you want to span more than just two octaves, you can change the overall pitch of the channel playing by using the octave command **O** followed by a number from 0 to 8. If you do not specify an octave (as in the example above), it defaults to 5 (the range containing middle C). The octave command remains in force for all notes following it until a new octave command is given.

This program lets you hear the same tune played in a higher octave (just add the **O7** to your earlier program):

```
10 a$= 'O7cfedafgCFEDAFGCC'
20 PLAY a$
```

Try changing the octave number progressively to hear the full pitch range which your ZX Spectrum Next's PSGs can produce.

Since each pitch range covers two octaves, two adjacent ranges overlap. For example, the high part of **O4** contains the low part of **O5** (see Figure below). The following diagram shows how you can create different notes using the **PLAY** octave command. As mentioned previously, the command **O** followed by a number from 0 to 7 sets the current PSG to a range of two octaves beginning with a C. The diagram shows the complete range of notes covered by **O3**, **O4**, and **O5**. Adjacent octave ranges overlap, so the same notes appear in the upper part of one range and the lower part of another. Individual notes within an octave range are set by using the letters **c** to **b** in lower case for the lower notes and **C** to **B** in capitals to give the notes in the upper octave. Placing a **#** before any note letter gives a sharp note; a **b** flattens it.

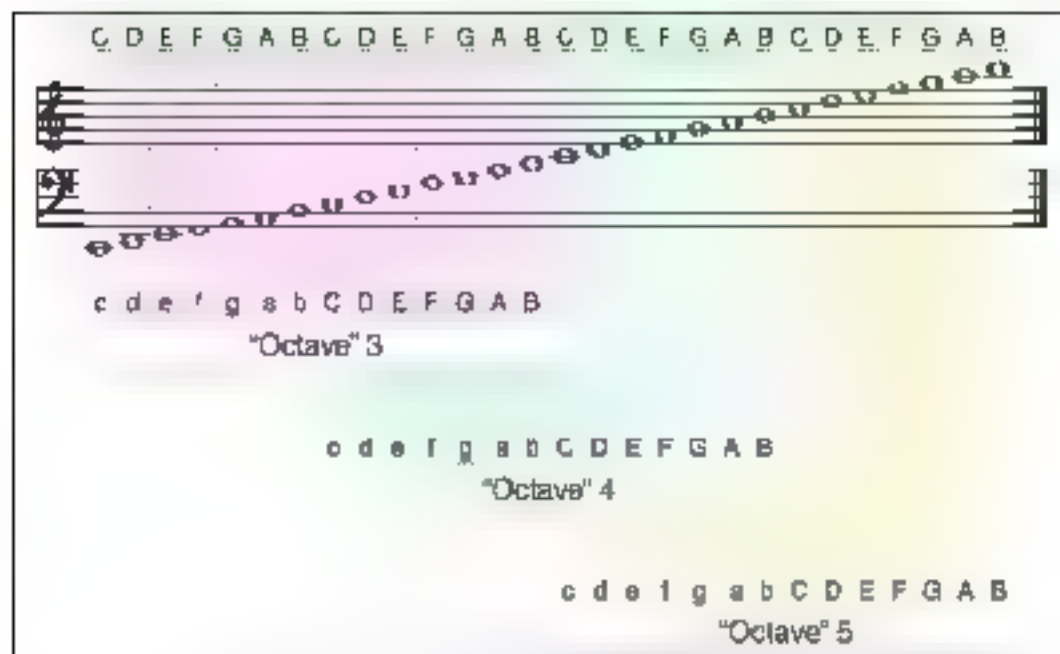


Fig. 18 Octaves and Pitch values for making music with **PLAY**

Note duration

If you do not specify the length of each note, they will all be played at the same length, as **xtotonefs**, as in the examples above. You can fix the length of any note or series of notes by prefixing it with a number from 1 to 12. This program lets you hear the different note du-

ration with numbers from 1 to 9 (there is a reason for the maximum number being 9 in this example as you will see in the table below)

```
10 a$="102030405060708090
20 PLAY a$
```

The **PLAY** command supports 9 standard musical durations: from a *semiquaver* (sixteenth note) to a *semibreve* (whole note) of the time signature. There are three extra duration values which denote *triple* notes (three notes played in the time normally used for what from a *triple* semiquaver (triple sixteenth) to a *triple* crotchet (triple quarter). While the first 9 values are set and apply to all the notes that follow a triplet duration value (10-12) only applies to the next 3 notes that will follow it in the string. For example:

```
10 PLAY "11ACE"
```

plays a *triple* quaver of A C and E. The following table lists the note duration values and their musical term equivalent:

| Value | Note name (Standard) | Note name (British) | Musical notation |
|-------|----------------------|---------------------|------------------|
| 1 | Sixteenth | Semiquaver | |
| 2 | Dotted sixteenth | Dotted semiquaver | |
| 3 | Eighth | Quarter | |
| 4 | Dotted eighth | Dotted Quarter | |
| 5 | Quarter | Crotchet | |
| 6 | Dotted Quarter | Dotted Crotchet | |
| 7 | Half | Minim | |
| 8 | Dotted Half | Dotted Minim | |
| 9 | Whole | Semibreve | |
| 10 | Triplet sixteenth | Triplet semiquaver | |
| | Triplet eighth | Triplet quarter | |
| 12 | Triplet quarter | Triplet crotchet | |

Table Note duration values

Additionally there is also the ability to insert moments of silence (or rests as they're called in music terminology) denoted by the ampersand symbol &. Rests last as long as the current note playing. For example:

```
10 PLAY "7A&B&C&D&E"
```

is five minims with equal (minim-length) silence durations between them.

Tied notes can be indicated by giving the two note durations connected by an *underscore* character `_`, and the note name: eg

```
10 PLAY "3_5A"
```

The second note duration you give will also apply to any following codes until you give another duration code.

The N Command

In some of the examples you will see the letter **N** used to introduce a series of notes within the string

```
PLAY 'O7N1CDE'
```

N is used in cases where two sets of numbers would otherwise clash. In the example above **O** is set to octave 7, then a series of notes is given, starting with the duration code 1. Without the **N** code, **NextBASIC** would read the octave code as 7 – obviously not what was intended.

Note volume

The overall volume of the sound is controlled by the volume setting of your display or amplifier. You can control, however, the volume of individual notes and phrases within the tune by using the **V** command. **V** followed by a number from 0 to 15 sets the notes to follow to a constant volume level. The lower the number, the quieter the sound, with **V0** being completely silent. **V0** is a useful way of stopping one channel playing while others continue. **V15** is the maximum possible value and will be used automatically by **NextBASIC** if you do not specify a level.

The low volumes are very quiet and you will normally use 10 to 15 unless you are outputting to an amplification system. Try running this program

```
10 A$="V10cdefgabCDEFGAB"
20 PLAY A$
```

Now try changing the number after the **V** to a new value to hear the difference.

Volume effects

Instead of you just setting each note to a fixed volume, **PLAY** also lets you change the volume of the sound while it's playing. For example, you can make a note start suddenly and then die away (like a piano), or make a sound effect rise and fall in volume (like a steam train).

This effect is controlled by the letter **W** which can be included in any of the strings controlled by the **PLAY** command. You must also include the letter **U** in each string where you want to use the effect. You cannot use it if the string already has a volume setting, if it contains a **V**. The volume command will override the effect.

The **W** must be followed by a number from 0 to 7 which controls how the sound builds up (called *attack*) or falls off (called *decay*). Table 13.1 that follows shows the full range of numbers and what they do together with a visual representation of the volume effect applied to the sound playing.

This program plays the same note with each effect in turn to let you hear what they sound like.

```
10 A$="UX1000W0C&W1C&J2C&
    W3C&W4C&W5C&W6C&W7C"
20 PLAY A$
```

Notice the **J** to turn on the effect, then the series of **W** numbers.

There is one other new command used here: the letter **X**. This can be followed by a number from 0 to 65535 to set the length of the sound effect – the larger the number, the longer the effect lasts.

The **X** command is not mandatory – you choose not to include one. **NextBASIC** will automatically choose the longest. In general, repetitive effects (**W4** to **W7**) are more effective

with short settings, eg **X300**. Single-shot effects (**W0** to **W3**) need a longer period, eg **X1000**. Try changing the value after **X** in the program above to hear the difference.

Tempo

The speed (tempo) at which a piece of music is played can be set with the command **T** followed by the number of *crotchet* beats per minute 'bpm' in the range 60 to 240. The command controls the speed at which all notes are played, but can only be included in channel A or PSG1 (the first string after the **PLAY** command) otherwise it is ignored, eg

```
10  a$='T160cdefg'
20  PLAY a$ 'T120CDEFG'
```

will play octave chords but at 160bpm as the second setting is ignored — no tempo is specified the music will be played at 120 bpm.

Repeated phrases

Any musical phrase can be repeated by enclosing the appropriate string or part of a string in parentheses. For example

```
10  PLAY 'abc(DEFG)'
```

will repeat the last four notes. If there is an unequal number of parentheses the phrase will be repeated back to the last parenthesis. If there is only a closing parenthesis the phrase will be repeated back to the beginning of the string. As an example

```
10  PLAY 'abcDEFG)'
```

will repeat all seven notes. Double closing parentheses

```
10  PLAY '02CEGA))'
```

will cause an infinite repeat. This is particularly useful for things like repetitive bass lines. To turn off an infinite repeat you will need to use the **H** command.





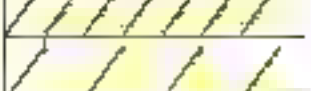



| Effect value | Visual Representation | Description |
|--------------|---|-----------------------|
| 0 |  | Decay then stop |
| 1 |  | Attack then stop |
| 2 |  | Decay then hold |
| 3 |  | Attack then hold |
| 4 |  | Repeated Decay |
| 5 |  | Repeated Attack |
| 6 |  | Repeated Attack-Decay |
| 7 |  | Repeated Delay-Attack |

Table 2: Volume effects values

The H command

An **H** included in any string immediately turns off the **PLAY** command. The main use of this is where you have an infinitely repeated bass line in one string. You can stop this at the end of the tune by putting an **H** on the end of the string which plays the melody.

Comments

You can include reminders and comments anywhere you like by using **'** marks. Anything written after a **'** will be ignored until the next **'** or the **'** at the end of the string is reached. For example

```
10 PLAY "abcDEFG chorus!aCEaDG"
```

Channel selection

The command **M** is used to select which of the three channels are in operation per PSG and whether these give noise or musical tones.

You can have a maximum of nine channels (three per PSG) in use at any one time but it does not matter whether they are all tone, all noise, or a mixture of both.

Your choice is entered with a number following the **M** worked out like this:

| | Tone Channels | | | Noise Channels | | |
|---------|---------------|---|---|----------------|----|----|
| Channel | A | B | C | A | B | C |
| Number | 1 | 2 | 4 | 8 | 16 | 32 |

Table 13 Channel audio type selection codes

Mark each channel you want to turn on, and note down its number from the table above. Then just add them together to get the code you should use after the **M**. For example, if you want to use tone channels **A**, **B** and **C** you add the numbers $1 + 2 + 4 = 7$ so you use the command **M7**. In the same way **M56** would turn on noise channels **A**, **B** and **C**.

Noise can be used on any channel, but the most wide-ranging frequencies are available in channel **A** for each PSG. For the best results, put your sound effects in the string which controls this channel for each PSG – the 4th and 7th string in other words, the 1st string per PSG after the **PLAY** command.

Stereo control

The **PLAY** commands **L**, **R** and **S** control the stereo image for each PSG. The first two reset the current PSG's audio output to Left and Right speakers respectively while the latter resets the Stereo image. Your ZX Spectrum Next is set up with **ABC stereo** (the default) normally channel **A** goes to the left speaker, **B** goes to left and right, and **C** goes to right.

Therefore if the **L** command is used, only channels **A** and **B** from the current PSG will be audible. Similarly if **R** is used, only channels **B** and **C** will be audible. Like the **M** command, the **L**, **R** and **S** commands need to be re-issued in the strings targeting each PSG.

Digital Audio

Your ZX Spectrum Next also contains hardware that can output digital audio, that is sound previously recorded digitally for reproduction in a similar manner to vinyl, CD, MP3 and MP3 players. There is no easy way to manipulate this hardware from NextBASIC so NextZXOS provides several *do!* commands. More on *do!* commands in Chapter 19.

¹ Comments are short fragments marking a block of code which are used at runtime. NextZXOS is a hybrid system, not normally available in NextBASIC to the user. Its commands were originally created for emuDOS, an Atari-style, real ZX Spectrum compatible Operating System which also works on the ZX Spectrum Next, and which NextZXOS has adopted as its emuDOS emulation layer. Most emuDOS third-party programs will work with NextZXOS and vice versa unless they use some special facility not covered by either the emuDOS emulation layer or the emuOS or NextOS (if available) support.

NextZXOS and alternatives, written by David Gahner and Kev Brady, that can be incorporated into your programs and which not only allow you to play any WAV file stored on SD Card media but also a plethora of digital audio formats.

Currently via *NextZXOS* you can play natively (you'll see why we explicitly mention it in a second) the following audio file types:

WAV

This is the standard audio format for most computers. The ZX Spectrum Next supports audio resolutions up to 32kHz. In order to playback a digital audio wave file type

```
wavp ay92 file.wav
```

where *file.wav* is the audio file you want to play. This can be accessed (like all other *NextZXOS* dot commands) from the 48k BASIC environment as well and fully incorporated into all your *NextBASIC* programs. You can find more information on how to access the digital audio hardware of your ZX Spectrum Next in *Chapter 22: I/O, QUT and the Next Registers*.

MOD

MODule files are one of the major standards for computer music and they comes from the Amiga and its ProTracker application. There are two ways you can play MOD files on the Next. You either use the dot command *nxmod* with your selection of *mod* file as an argument, for example

```
nxmod song2.mod
```

or you can use the native application **NXModPlayer**. This can be found under *c:/apps/audio/NXModPlayer*, accessible either via the Browser (See relevant section on the Browser in the next Chapter) or via the commands

```
as = 'c:/apps/audio/NXMod/nxmod  
play nex'  
.nexload as
```

PT3

PT3 is one of the de facto standards for AY chip music and the ZX Spectrum Next supports playback of up to 6 channel audio in two ways. First is via the dot command *playpt3* with the *pt3* filename as an argument

```
p aypt3 onlygoj.pt3
```

or via the application **NextSID**. This is a rather special application as it not only allows you to playback pt3 music files but also to apply SID-like effects to the channels. **NextSID** can be found under *c:/apps/audio/NextSID*. As with **NXModPlayer** above you can either start it via the Browser or via the commands

```
as='c:/apps/audio/NextSID/nextsid nex  
.nexload as
```

Note that you will need a mouse installed

Using the PI accelerator for audio

If you have the *Accelerated* version of the ZX Spectrum Next or have a *Raspberry Pi Zero* installed on your board, then you have more options available audio-wise. These include (but are not limited to) playback of:

- Commodore 64 SID files
- "Tracker" MOD files

- Aiacr .ST SDH files
- MP3 files
- High definition wav files

and many many more

The way the system works is as follows. The ZX Spectrum Next communicates with the Accelerator via its secondary UART and sends commands and audio files to the specialised SUPERvision software that is running on the Raspberry Pi Zero. The Pi Zero in turn interprets these files and reproduces the audio contained therein via it's GPIO port onto the ZX Spectrum Next I2S port which in turn mixes it with the rest of it's audio output and redirects it to whichever output you have available. In essence when it comes to playback, the ZX Spectrum Next is considered a sound card where the accelerator is concerned and two extra DACs where the ZX Spectrum is concerned. As a consequence you can have Digital Audio on the ZX Spectrum Next, all three PSGs playing AND Digital Audio on the Pi Zero³ all playing simultaneously!

To use the Pi audio facilities you need to first enable the secondary UART and set it to the accelerator. In *NextBASIC* or the *Command Line* you must type

```
CD "c:/apps/pi"
```

and press ENTER. Then type

```
LOAD "pi.bas"
```

You'll get a message stating **9 STOP** statement, 50 indicating the system is now ready to play audio using the Pi Zero. Feel free to poke about the listing of the **PI BAS** program as it shows you the usage of *Next* registers (see *Chapter 22* for more).

Playing audio files requires a dot command called **pisend** which you can find in **c:/dot**, which serves a two-fold purpose to send files to the Pi Zero's temporary storage and send the appropriate command for it to play. Thankfully D. Ammon-Soutter and David Gaphier maintain ours of *NextPI2*⁴ and **pisend** respectively have packaged all this nicely into little *NextBASIC* programs (located in **c:/nextzxos/**) which you can either call directly or via the Browser by selecting a filetype already registered. Currently registered filetypes include **SID** **MOD** **XM** **TXZ** and **SDH**.

To illustrate how this works we shall attempt to play an Aiacr .SDH file. Assuming you have a .SDH file named **warhawk.sdh** (search for it and download it on the internet. It's freely available) on the root of your SD card, playing it is as simple as

```
LOAD "c:/nextzxos/sndplay.bas"
f$="c:/warhawk.sdh" GO TO 10
```

The screen will read **Playing: c:/warhawk.sdh** and the music will start playing from your speakers.

External Audio Output

If you are interested in doing more with sound from the ZX Spectrum Next, like hearing the sound that **BEEP** and **PLAY** make on something other than the usually limited audio of your display, you will find that the audio signal is also present on the Audio Out socket on the back of the machine. You may use this to connect to a pair of headphones or a higher quality amplifier. Note that this will not disrupt audio reproduction on the digital display cable, therefore you may want to turn down the volume on your display before plugging an

³ I2S is Universal Asynchronous Receiver-Transmitter is a hardware device that exchanges data sequentially between two systems. In our case this is done between the ZX Spectrum Next hardware and the Pi Zero accelerator via its GPIO port.

⁴ *NextPI2* is a software emulator that runs on the Raspberry Pi Zero.

⁵ *NextPI2* is the operating system running on the Pi Zero, an emulator that purposefully is to support the Next

external audio reproduction device. Note also that there is no volume control for the Audio Out socket so you should take that into account when using headphones or an amplifier.

Notes

TZX files are perfect ZX Spectrum tape images. Due to them being compressed, they require a much more powerful CPU than the Z80A present in the Spectrum Next in order to be decompressed to their original tape audio stream. While not audio in the strict sense with us discussing in this chapter how to use the audio subsystem in code loaded on the ZX Spectrum Next, side and as such they are covered here.

Exercises

- 1 Rewrite the Mahler program so that it uses **FOR** loops to repeat the bars.
- 2 Program the computer so that it plays not only the funeral march but also the rest of Mahler's first symphony.
- 3 Repeat exercises 1 and 2 above by utilising **PLAY** instead of **BEEP**.

Chapter 19 NextZXOS and alternatives

Guide to NextZXOS

Until now, we have been talking about *NextBASIC*, the programming language with which you 'talk' to your ZX Spectrum Next, and get it to do things. Underneath *NextBASIC* now ever lurks another program, one that allows your computer to communicate with the hardware devices connected to it and the world at large. It manages your computer's memory, makes sure your data is safe and accurate, that your programs behave as intended by their programmers and performs important housekeeping of your storage devices. This program is called an *operating system* and in the ZX Spectrum Next's case it is called *NextZXOS*.

NextZXOS, written by Garry Lancaster, is the direct successor to his +1e/*IDE/DOS*, which in turn comes directly from the first proper Sinclair ZX Spectrum operating system called +3DOS which first appeared on the ZX Spectrum +3.

NextZXOS main features

NextZXOS extends +3DOS, 3e and *IDE/DOS* and features the following:

- FAT16 and FAT32 support for industry standard compatibility with mass storage devices while retaining *IDE/DOS* +3DOS compatibility for a full range of storage devices
- Long File Name (LFN) support
- Proper subfolders/subdirectories
- Memory Management facilities
- *virtual container file systems in disk and tape images*²
- installable device drivers
- Menu-driven file manager with extensible filetype associations/launchers
- esxDOS emulation layer for interoperability across ZX Spectrum compatible machines and extended *dot* command support
- Automatic execution of software on boot
- Command-line interface
- *Streaming* support
- *Virtual* memory support (swap partitions)
- Timekeeping facilities
- Availability of disk and file management even on legacy (via *dot* commands 48K modes)
- Increased compatibility with previous models of ZX family of computers³
- Support for a variety of snapshot formats
- Multi-lingual and multi-font capabilities
- Extended windowing facilities
- Increased speed of operation compared to the previous versions
- Proper CP/M-3 compatibility

Unlike other operating systems, *NextZXOS* tightly integrates with the in-built programming language *NextBASIC*. In the point that it can be mistaken as being part of it, in reality however, *NextZXOS* provides two rich APIs, one being the native *NextZXOS* API and the other

² *Virtual* Name support means that a filename under *NextZXOS* can be up to 255 characters long as opposed to the earlier 80 characters. *Virtual* directories, containers, images and more. The capability is not given to further folders, but also to work with character long, longer file and folder names help with the organisation of archives as it is easier to use much less verbose names.

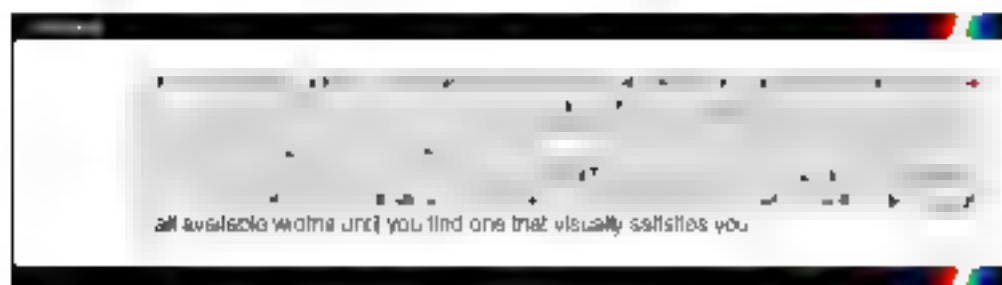
³ A major feature of the system, disk images via snapshots, copy other contents of a mass storage medium, together with the single use, for example when used to be an entire floppy disk, can be represented by one file under *NextZXOS* and accessed with traditional disk and file management commands once they are attached (mounted) by the operating system.

³ *NextZXOS* is compatible via emulation provided by Paul Farrow with ZX80 and ZX8, while also being more powerful than its predecessor with a new 32-bit architecture, the *48K and disk* steps of ZX Spectrum capabilities.

⁴ CP/M is an older operating system for personal computers with a vast library of software.

is) and their place is taken by the aforementioned dot commands

comfortable that way



Files, Drives, Partitions and Disks

NextBASIC in turn, can be extended to handle more file types

tion of files grows from a few tens to hundreds or thousands

Working with files

Working with files

LOADED them by using two commands **SAVE** and **LOAD**

type *word* is specified. It must be preceded by a dot. Unlike some other BASICs, NextBASIC does not automatically allocate a type to files if one is not specified.

You may find it useful to add your own types: a popular convention is to use **BAS** to identify NextBASIC file types and **BIN** or **COD** to identify machine code file types.

NextBASIC already understands a number of popular types. Going to `c:\nextzxos` with the browser, placing the cursor over `browser.cfg` and pressing **ENTER** will return the most commonly used ones together with the action that will be taken when the Browser launches them.

The characters ***** and **?** are called *wildcards* and have a special meaning to NextZXOS. They're used to substitute ranges of characters or specific characters in filenames and folders. We'll see why this is particularly useful further below.

The dot character `.` also has a special meaning according to how many we use. If we use one `.` it means *this folder* and if we use two `..` it means the *folder one level up*. Keep this information in mind as it will prove very useful in the examples we'll encounter.

The following are some examples of valid filenames:

- `z`
- `squares`
- `m picture bin`
- `a:fred`
- `13a.hello`
- `OM-CAPITALS`
- `file name`
- `test.bas`
- `philip`
- `glass.mus`
- `a:a a`
- `c:\nextzxos\browser.cfg`
- `c:\nextzxos\browser.cfg`
- `7.dubious`

while the filenames below are illegal and attempting to use them will produce an error:

- `<> += &` (must not contain any of these characters)
- `*test` (cannot contain an asterisk)
- `!e?sl` (cannot contain a question mark)

Note that in the list above we've made two assumptions regarding valid filenames, and these are that drive names `a:` and `m:` are *virtual disks* and the *RAMdisk* respectively. `/` `se` areas are acceptable parts of filenames ONLY if the drive's filesystem allows them, otherwise you will get an error.

With that information in hand, let's start examining below the main commands for working with files.

LOAD

LOAD as its name implies retrieves a file from a drive and puts it *loads* it in the computer's memory. Depending on how it was saved (in the case of NextBASIC programs, or named (in the case of machine code software) it may also execute it as well. It takes the form:

LOAD *filespec* [**MODIFIER** *options*],

where *filespec* is a filename as described in the previous section, followed by an optional **MODIFIER** directive (**SCREEN\$**, **LAYER**, **CODE**, **DATA**, **INT**, **INPLT** or **BANK**) which in turn may have optional parameters.


```
20 INK 3. PAPER 8 PRINT
    Hello World!"
30 SAVE "test.scr" SCREENS
```

dr. u. b. JUN. Yip, who has been a member of the committee since 1994, said that the committee will continue to work with the government to improve the system.

Now type

```
CL5 LOAD "testscr" SCREEN#
```

line 30 and replace SCREENS with LAYER so it reads

30 SALE "test scr" LAYER

and **RUN** it again. Then give the following:

```
CLS LOAD "test.scr" SCREEN$ PRINT AT
2 2, "Press Any Key" PAUSE @ CLS LOAD
test.scr" LAYER
```

at the ENTER key, you will see a "LOAD" message on the screen with
 immediate plot. To plot, press 3 key. Before the END key, you will see the message
 "thing".

```
LAYER 2,1 LOAD "test.scn" SCREEN# PAUSE
0 LAYER 2,0 LAYER 0
```

This will produce a blank screen waiting for a keypress which when pressed will give its location in the screen and previously saved locally to the file at the next keypress.

```
LAYER 2,1 LOAD "test.scn" LAYER PAUSE
0 LAYER 2,0 LAYER 0
```

[illegible]

LAYER 0 LOAD 'test.scr' CODE

TYPE dialog will be created by the period 'Hello World' screen we generated previously. To expand a bit on this first type

NEL

After pressing **ENTER**, you'll be greeted by the *NextZXOS Startup menu*. Select *NextBASIC* and rewrite the line above by adding **16384,6144** at the end after the **CODE** to read:

```
LAYER 0 LOAD 'test.scr' CODE 16384,6144
```

Amazingly the **Hello World** message reappears but this time colourless. Adding the two numbers after **CODE** instructed the computer to load the file at address **6384** which is the start of Layer 0's graphics memory but at a smaller length than the actual file we stored, removing all the colour attribute information. Attempting to store a longer length than the size of the file we're loading, the computer will return an **End of file 0.1** message. Note here that doing just that is not a good practice and we should be using **LOAD BANK**, **LOAD SCREEN\$** and **LOAD LAYER** to load data into graphic memory.

As we saw in Chapter 1, one of the most tedious aspects of programming is to prepare arrays. They can involve endless typing via data statements and use a lot of program space which could otherwise be used for actual program logic. Thankfully *NextBASIC* gives us the option after we've prepared an array, to save it to a file to be retrieved later, saving us both time and code memory. To load such prepared arrays we need to use the **LOAD** modifier **DATA**. This takes the form

LOAD filespec DATA arrayname()

to load for example the array **b()** from Chapter 2 assuming we have already saved it as **b-array.dat** we'd only need to type

```
LOAD "b-array.dat" DATA b()
```

This would find if any other array named **b()** was already stored in the computer's memory erase it and replace it with the information provided in the file.

We can only load string and floating point arrays. Also of note is that the parentheses after the array name cannot be omitted.

Integer arrays cannot be loaded or saved with the **DATA** modifier. For that we need the following modifier **INT**.

LOAD filespec INT

will load previously saved integer arrays and variables. All arrays and integer variables will be initialised prior to loading and replaced with what is in the saved file. Additionally the **INPUT** modifier

LOAD filespec INPUT

will load a previously saved definition of the keyboard joystick. See Chapter 4 and the **INPUT** function for more.

The final **LOAD** modifier **BANK** should be looked upon as a variant of the **CODE** modifier as it basically loads raw data into memory in the bank number offset. Set bank and length (in bytes) we specify very much like **CODE** does. This takes the form

LOAD filespec BANK number [offset] [length]

Keeping with the example we have been using try

```
LOAD "test.scr" BANK 5
```

will load and display the exact same screen with the main difference that it will put it in offset 0 or bank 5. For reasons that will become clear in Chapter 23 this is exactly the same location as the one we used with **SCREEN\$** and therefore if you slightly modify the command to be

```
LOAD "test.scr" BANK 5, 0, 6144
```


as previously, the file will appear colourless. When using **BANK** as a **LOAD** modifier, we need to remember that NextBASIC and NextZKOS do not care what type of data is being loaded. As such the **BANK** modifier is also used to load NextBASIC programs that make the use of banks. More about that below when we examine **SAVE**.

SAVE

Our computer's memory lacks permanence: whatever is stored inside it during operation disappears when we turn the power off. We need some means to store the information into a medium that can hold it even when the power is off. This comes in the form of the **SAVE** keyword.

It follows the exact syntax of **LOAD** that we examined in the previous section and uses the same modifiers and parameters with an additional **LINE** modifier. There are a few differences from **LOAD** in behaviour however and we'll examine these immediately. Typing

```
SAVE ""
```

will produce an **F Invalid file name, 0:1** error even when our default drive is **T** (tape). That's simply because even on a tape, files **NEED** to be named, otherwise we wouldn't be able to identify them.

As with **LOAD**, setting the filespec to a drive name (for example **c**) will switch all NextBASIC file retrieval and storage operations to that drive from that point forward so for example

```
SAVE "m."
```

will make drive **m** the default drive and won't actually store any information anywhere.

As we saw in examples in the previous section, **SAVE filespec** without a modifier (assuming filespec is a string specifying more than just a drive name) will save the NextBASIC program currently in memory into the default drive or the drive/folder we specify. If however this filename already exists in the location specified, NextZKOS will first create a backup file made up from the original filename and then append the type **bak** to it.

We will have to skip ahead again to see the results of our operations by using **CAT** (for CAtalogue) so let's quickly do some typing.

```
SAVE "c
```

```
10 PRINT "Hello"
```

and then

```
SAVE "hello.bas"
```

followed by

```
CAT "hello*.*"
```

(Never mind what the ***.*** means, we'll examine that later)
Your screen will display the following

```
hello bas                                1K
960M free
```

Now perform the save again, again followed by **CAT "hello*.*"** and you'll see

```
hello bas                                1K
hello bas bak                            1K
```


980M free

before we discuss what has happened, make a small modification to the program, for example add an exclamation mark after World on line 10 and do another save, a bit different this time

```
SAVE "hello"
```

and follow it by `CAT "hello" **`. Now you'll see

```
hello                1K
hello.bas            1K
hello.bas.bak        1K
```

980M free

Repeat the last save command one more time and then do `CAT "hello" **` again. The screen now shows

```
hello                1K
hello.bak            1K
hello.bas            1K
hello.bas.bak        1K
```

980M free

you, however, had started with a `SAVE "m"` thus reverting the default drive to the RAMdisk, everything would have been a bit different. First by not displaying a `hello.bas.bak` and now after the entire series of commands `CAT` would have returned

```
HELLO                1K
HELLO.BAK            1K
HELLO.BAS            1K
```

59K free

so, why the difference? Let's take it from the beginning. We initially saved a NextBASIC program that was named `hello.bas`, then once we saved it again, the file with the same name on the drive had a `.bak` type appended to it. Then we saved the same program with a name without a type. In the second case, since we were trying to save to a FAT filesystem, the RAMdisk, NextZXOS can only use 8 + 3 character filenames unlike the FAT filesystem, that can have very long filenames. So in the second case, instead of appending the `.bak` type to the original `hello.bas` file, it stripped the `.bas` type and replaced it with `.bak`. What followed is that we tried to save the same name without type but now NextZXOS had a decision to make, which filename with `.bak` type to keep? As you could easily find out by `LOAD`ing back the `hello.bak` file, the last version saved is the one retained. Your `PRINT` statement would be the one with the exclamation mark and not the one without.

This example makes an important point, that due to the disparate types of filesystems NextZXOS can handle, the auto backup feature provided is nice but it's not a panacea, so do not rely on it exclusively and instead name your files explicitly!

A slight variation on the `SAVE` command as it deals with NextBASIC programs is that you can add the `LINE` modifier with either a numerical parameter or a label name after it. For example saving the program above with

```
SAVE "hello.bas" LINE 10
```

and then doing

```
LOAD "hello.bas"
```


will load `AN()` start the program at line 10 or at the specified label² which will then print **Hello** on your screen. As a matter of fact you can use even non-existing line numbers when saving. `LOAD` will go to the first available line after the one you entered if that doesn't exist in your program and attempt to run from there. If the line number you entered is higher than the last line number in your program, `LOAD` will just not execute the program as simply loading it as the `LIN` modifier was never specified. `SAVE filespec LIN` number will NOT accept a number greater than 65535 however and it will return a **Integer out of range** error if such a value is supplied for number or **0:1 No Label** error if the label doesn't exist.

It is noteworthy that a particular type is not forced upon the file when using `SAVE` so a `NextBASIC` file for example will not automatically carry the type `bas`. That being said, as we saw earlier, a standard set of types is known to the NextZXOS browser. These help it automatically launch files using the appropriate commands. It is therefore a good idea to either adopt these or modify the ones known to NextZXOS to be the ones you prefer. Remember however that every time you update `System/Next™` the known associations to file types are being overwritten with the default ones, so always keep a backup of the browser `cfg` file located in `c:/nextzxos/` if you indeed make these changes.

As we saw earlier, storing screens requires the use of either the `SCREEN$` (for Layer 0) or the `LAYER` (for all other layers) modifier directives. From our examples, you may have already assumed that the `LAYER` modifier can also be substituted by the `BANK` or `CODE` modifiers. While this is true for Layers 0 and 1, there's no functional way this can be done for Layer 2 with `CODE` or `BANK` as the latter occupies more than one banks and `CODE` only works within the main memory map.

The most compatible way to save screens is therefore the use of the `LAYER` modifier directive as follows:

```
LAYER desired_layer
< statements generating graphical content >
SAVE filename.ext LAYER
```

Remember that you must already be in the layer that you intend to save before initiating a `SAVE LAYER` command. Also, as you can find from looking at browser `cfg`, NextZXOS already recognises some types as belonging to a specific layer screen file. The table below lists them in order:

| Type/Extension | Layer |
|----------------|--------------------|
| SCR | JLA (Layer 0) |
| SL0 | LoRes (Layer 0) |
| SL1 | HiRes (Layer 1) |
| SLC | HiColour (Layer 2) |
| SL2 | Layer 2 |

Table 14: Automatically recognisable screen file types

By this time and given the time we spent discussing the `CODE` modifier, you've probably figured out that it's not reserved for machine code programs and instead will save or load the raw data that's located in the memory address you specify, whether this is graphics, machine code, a `NextBASIC` program, variables, `NextZXOS` system variables or just random numbers or even nothing (0s).

Like its `LOAD` equivalent, `SAVE CODE` requires both parameters: that is a legal address and *valid* length. It takes the form:

```
SAVE filespec CODE start address length
```

² Although it can be anywhere in a file, for `SAVE AN()` and `LOAD AN()` to work, the address must be the address of the label in the line where the label is located and not at the location of the label.

where *start* address can be any number from 0 to 65535 and *length* any number from 1 to 65535 and the sum of these should not exceed 65536. **CODE** as discussed works only in the main memory or, alternatively, in the main memory map and, or in the rest of the memory, we should use the **BANK** modifier. The main difference is that **BANK** is only 16K in size (thus accepting a maximum of 16384 as offset² and *length* **BANK** can be used without an offset or length, but once an offset has been specified, the length parameter is required). Saving the contents of a bank takes the form

SAVE filespec BANK number [offset, length]

For NextBASIC programs that make the use of memory banks (as we'll see in Chapter 23), apart from the main program that can be saved with a simple **SAVE** command, you also need to save all the banks that contain parts of the program. It is therefore imperative to use **SAVE BANK** on its own (without offset information) to make sure that all the NextBASIC parts are saved. As you will also see it's good practice to also assign banks when writing a NextBASIC program using variables so when you're loading them back you don't have to literally assign specific bank numbers as these can be reused by NextZKOS or a machine code program already in memory.

We already saw how we can use **LOAD** to load arrays into NextBASIC without having to enter complex **DATA** statements that have the potential of making our program hard to read. We **SAVE** arrays by using the **DATA** modifier followed by the array name, including parentheses, we wish to store for later usage. A few things we need to note are:

We cannot use a non-dimensioned array in our **SAVE** statement. For example if we do

```
SAVE "data" DATA a()
```

we're more than likely to receive a **2 Variable not found 0-1 error**. Writing something like this

```
DIM a (3) SAVE "data" DATA a()
```

however will save happily.

An already dimensioned array can be saved using a direct NextBASIC command or as part of a program or a saved array loaded using the command line in a direct NextBASIC command will NOT be available from your program unless it's loaded explicitly from it. Let's illustrate this point by writing the following little program:

```
10 DIM a (30)
20 FOR f=1 TO 30
30 LET a(f) = 30 / f
40 NEXT f
50 SAVE "data" DATA a()
```

RUN the program and then type **NEW** to restart NextBASIC. Then type the following program:

```
10 FOR f=1 TO 30
20 PRINT a(f)
30 NEXT f
```

² In early NextBASIC, in order to remain compatible with earlier versions of Sinclair BASIC, allows an invalid integer but here, as both address and length have to be valid, we must specify a valid length. The address can be either an absolute or a bank address, so make sure you verify that the locations whose storage are inside the actual memory map using the term offset to mean absolute address and address to mean bank address. A bank as it can mean 'segment' in the memory map, locations within a bank always start at 0 and that's common to all banks.

you RUN the program you'll get a **2 variable not found 201** error reminding that at line 20 NextBASIC has no idea what **a** means. Now without erasing the program give the following series of commands:

```
LOAD "data" DATA a() FOR d=1 TO 30 PRINT
a(d) NEXT d
```

You'll get the same series of numbers you stored with the previous program before you typed **NEW** or so on. If you however attempt to RUN the program you just typed, the **2 Variable not found 201** error will persist. In order to fix this you will need to add the following line:

```
1 LOAD "data" DATA a(),
```

which will produce the same effect as the direct command you gave earlier. You do not need to DIMension the array as **LOAD** will do that for you. It is also useful to note that if doesn't match with array's size you saved before, when you load the same data back you can assign it to any available array. So you could theoretically **SAVE "data" DATA a()** and **LOAD data DATA b()**. The only thing you need to remember is that the array you must match the data saved, otherwise you will receive a **b Wrong file type 01** error.

Using the **INT** modifier with **SAVE** will store a snapshot of all your integer variables and arrays. All 24 integer variables and 24 integer arrays are saved regardless of whether or not data or not.

Finally the **INPUT** modifier used with **SAVE** stores your current keyboard layout key as segments.

VERIFY

When storing data on tape in order to make sure what the program or raw data that you've saved is accurate, NextZXOS provides NextBASIC with the **VERIFY** command. On media other than a tape, **VERIFY** has no effect unless it's used in conjunction with a drive name in which case it will act like its **LOAD** and **SAVE** counterparts switching the default drive to the one specified. In every case, if used in tape mode, **VERIFY** will return **0 OK 01**. **VERIFY** follows the same syntax as **SAVE** except for the **LINE** modifier. Assuming you have a tape deck attached to your ZX Spectrum Next and having the Hello World program we typed a little earlier, save the program into tape by giving:

```
SAVE "t" SAVE "hello.bas"
```

Now we will try to make sure that the program was saved to tape properly by doing the following:

1. Rewind the tape to just before the point at which you saved the program
2. Type
VERIFY "hello.bas"
3. Play the tape. The border will alternate between red and cyan until NextZXOS finds the program that you specified. Then you will see the same pattern as you did when you saved it, and after a pause between patterns the message **Program hello.bas** will be displayed on the screen. When NextZXOS is searching for something on tape, it displays the name of everything it comes across. After the pattern has appeared, you see the report **0 OK**, then your program is safely stored on tape and you can skip onto the next section. Otherwise, something has gone wrong. Take the following steps to find out what:

If no program name has not been displayed, then either the program was not saved properly in the first place or it was but was not read back properly. You need to find out which of the two is true. If it was saved properly, rewind the tape to just before the point at which you saved the program, then play it back while watching the video display.

The red and cyan lead-in should produce a clear, steady high pitched note, while the blue and yellow information part gives a much harsher screech.

If you do not hear these noises, then the program was probably not saved. Check that you were not trying to save the program onto the plastic leader at the beginning of the tape. When you have checked this, try saving again.

If you can hear the sounds as described, then **SAVE** was probably alright and your problem is with reading back.

It could be that you mistyped the program name when you saved it (in which case when NextZXOS finds the program on the tape it will display the mistyped name on the screen). On the other hand, perhaps you mistyped the program name when you loaded it, in which case NextZXOS will ignore the correctly saved program and carry on looking for the wrong name, flashing red and cyan as it goes.

If there is a genuine mistake on the tape, then NextZXOS will display an **R Tape loading error** which means in this case that it failed to verify the program. Note that a slight fault in the tape itself (which might be almost inaudible with music) can wreak havoc with a computer program. Try saving the program again, perhaps on a different part of the tape, or a different tape altogether.

MERGE

Many programmers like to store parts of their programs or special subroutines they want to use again and again, thus building *libraries* of code. Normally a subroutine will be part of a larger program but what if it could be used anew on a different kind of program? Normally you would have to load the entire program into memory, edit out the parts you do not need and then proceed to write the rest of the new program only leaving the part that you want to reuse intact. Similarly, there may be someone that only wants a routine to be used once into their program (for example during initialisation) and then exchange that space for another routine that performs a completely different task. The answer to both these issues is the **MERGE** command. **MERGE** is used in the same way as **LOAD** with the difference that it doesn't clear what's in memory already and does not erase the program's variables and instead only replaces lines that already exist. To illustrate this point consider this little program:

```
10 PRINT 'Part 1'
20 PRINT 'Part 2'
30 PRINT 'Part 3a'
50 PRINT 'Part 5'
```

Now save the program by giving

```
SAVE "part-a.bas"
```

and then give the command:

```
NEL
```

After you re-enter NextBASIC and type **LIST** you will see there's no program in memory. At that point type

```
30 PRINT 'Part 3'
40 PRINT 'Part 4'
60 PRINT 'Part 6'
```

Now save this program also by giving

```
SAVE "part.b.bas"
```


| | |
|--|---|
| <pre> a b c a b c a b c a b c </pre> | <p>Any filename with any type that ends in the letter a</p> <p>Any filename starting with a with any type</p> <p>Any three letter filename ending with the letter a with a type having a b as second letter (for example dba.bbf)</p> |
| <pre> a b c d e f a b c d e f a b c d e f a b c d e f </pre> | <ul style="list-style-type: none"> * not the last character in the Name field * not the last character in the Name field * not the last character in the Type field |

our files

Filesystems

obviously affects some of the features we'll examine below

of large media like hard disks

Partitions

or FAT32 – using FAT as a portmanteau term is acceptable use

age devices

Storage devices and disks

ern numbers, and each partition on each disk (if a partition exists) is assigned a number in turn. *Virtual disks* on the other hand do not have device numbers as they don't physically exist however both require a *driver*, that is a small program that sits between the disk and NextZXOS and translates each device's individual characteristics into the common set of controls that NextZXOS understands. That alone however is not enough. NextZXOS needs to assign a drive to each partition on a disk (or in the cases of *virtual disks* and the *RAMdisk* to the disk itself). As it comes with your **System/Next™** distribution, NextZXOS knows three types of physical disks: SD Cards, the *RAMdisk* and floppy disks and two types of *virtual disks*: 1 floppy disk images and 100DOS hard disk images. It also knows *virtual* and *physical* tapes both addressable via the reserved drive t. Physical disk device numbers start at 0 and are assigned according to the table that follows:

| Device Number | Description |
|---------------|--|
| 0 | All IDEDOS partitions on the first SD drive |
| 1 | All IDEDOS partitions on the second SD drive |
| 2 | Reserved for First Floppy Disk drive |
| 3 | Reserved for Second Floppy Disk drive |
| 4 | RAMdisk |
| 5 | All FAT partitions on the first SD drive |
| 6 | All FAT partitions on the second SD drive |

Table 5 Device Number assignments

On an unexpanded ZX Spectrum Next with an unmodified distribution of NextZXOS, the first used number is 4 which is the *RAMdisk* and the second is 5 as **System/Next™** comes on an SD card containing only a single FAT partition. As seen on the table above, device numbers 2 and 3 refer to floppy disk drives, not yet supported by NextZXOS.

Mounting

In order for NextZXOS and NextBASIC to know how to access a partition or disk (be it physical or virtual) this partition/disk has to be *mounted*. That is the process where a partition on a device gets attached to a drive. If freshly installed, NextZXOS will automatically mount two drives: drives c and m, the first being device 5 partition 1, in other words the **System/Next™** distribution's SD card plugged into the first SD reader of the system, and the second one being device 4, the *RAMdisk*. On an initialised CPM distribution (as we'll see further below) one more drive will be mounted and that's drive a, assigned to `cpm-a.p3d` located inside `c:\nextzxos\`.

Generally speaking, if there are more than one *FAT partitions* detected on the SD card, they will be automatically mapped to drives c onwards on startup.

Finally, any files located inside the `c:\nextzxos\` directory, are mapped to the appropriate drives (if the drive in question has not already been mapped) if they are named as follows and are valid +3DOS partition images:

DRV-A.P3D
DRV-B.P3D

DRV-P.P3D
CPM-A.P3D
CPM-B.P3D
()
CPM-P.P3D

Virtual images named **DRV-x.P3D** where x is a letter from a to p have precedence over *virtual images* named **CPM-x.P3D** so in the presence of both, the **DRV-x** variant will be mounted. Apart from the auto-mounting procedures described above, we can also manually mount partitions and disks. This will be covered a bit further below at its own section.

With all this information at hand, we can now proceed to examine NextZXOS facilities by asking:

Drive cataloguing

It's obvious that simply remembering a file's name and **LOAD**ing it is not possible after the first few files, so we need a command that can help us see which files are stored on a drive. This command is **CAT** from **CAT**alogue, and its syntax is as follows:

CAT [**S**] **#n** [*filespec*] [**EXP**]

where **S** is a switch instructing the file list produced to use the short (8+3) format, **#n** is a NextZXOS stream for the output of **CAT** to be redirected to, *filespec* follows the conventions described in the *libraries* section earlier, and the modifier **EXP** produces an expanded listing with more information about the files being listed. All **CAT** parameters are optional and by itself **CAT** will produce a listing of the default drive which can be set in the same manner as with **LOAD**, **SAVE**, etc. Try the following:

```
LOAD 0
CAT
```

You will receive the following on your screen:

```
No files found
62K free
```

```
0 OK, 0 1
```

Congratulations: you just listed the contents of the *RAMdisk*. Sadly, it's empty! Now type

```
LOAD "c"
CAT
```

Your display now will look similar to this:

```
DEMOS          <DIR>
DOCS           <DIR>
DOT            <DIR>
GAMES         <DIR>
MACHINES      <DIR>
NEXTZXOS      <DIR>
RPI           <DIR>
SRC           <DIR>
SYS          <DIR>
TMP          <DIR>
TOOLS        <DIR>
LICENSE.MD   6K
README.MD    2K
TBLBLUE.FJ   168K
TBLBLUE.TBU  468K

1687M free

0 OK, 0 1
```

which is a slightly modified listing of the contents of the *root folder* * of your **System/Next**™ distribution. Now type

```
CAT EXP
```

* In filesystems other than **ILUOS** and **ILUOS** that use user files, files are organized in an inverted way: an *see* or *sub* is contained in *folders* or *directories* and a *look* is the whole name, which is a *bridge* (or *path*) between the *see* and the *look*. The *see* is called the *main* folder or *root* directory.

Your display now will look similar to this:

```

CORES
2019 09 02 01 01
DEMOS
2019 09 02 01 01
LICENSE.MD
2019 09 02 00 07 5243
README.MD
2019 09 02 00 07 1427
TBBLUE FU
2019 09 02 00 07 172032
TBBLUE.TBL
2019 09 02 00 07 425543

```

You can immediately notice two things. First, the addition of a column made from four characters at the rightmost side of the screen and secondly that every entry now occupies two lines with the second containing a date and a number (not in all cases). Let's start from the second line. Two types of information is available here: when the file or folder was created and what's its size (in bytes). The first line is the file itself or the folder while the rightmost column describes the file's attributes. The d you can see in some entries is the directory attribute which designates a folder. Folders as far as the filesystem is concerned are special files without size. In the shorter form of CAT we saw previously, this is displayed as <DIR>. There are many more attributes to examine which we will look at later.

You may have noticed that the display gets very cluttered when using the EXP mode: especially if there are a lot of files with long names as the screen normally fits only 32 columns. If you follow the note in the beginning of this chapter and use 64 or 85 column modes you'll see the situation improves. Switch to 64 column or 85 column mode, rerun CAT EXP and you will get something similar to this:

| | | | | | | | | |
|---------------|---|------|----|----|----|----|----|--------|
| PERIOD | d | 20 | 9 | 10 | 22 | 20 | 28 | |
| DDCS | d | 20 | 9 | 0 | 22 | 20 | 28 | |
| DO | d | 20 | 9 | 0 | 22 | 20 | 28 | |
| GAMES | d | 20 | 9 | 0 | 22 | 20 | 28 | |
| MARCMES | d | 20 | 9 | 0 | 22 | 20 | 28 | |
| NEE ZXOS | d | 20 | 9 | 0 | 22 | 20 | 28 | |
| RFI | d | 20 | 9 | 0 | 22 | 20 | 28 | |
| CRG | d | 20 | 9 | 0 | 22 | 20 | 28 | |
| TEKS | d | 20 | 9 | 0 | 22 | 20 | 28 | |
| FMP | d | 20 | 9 | 0 | 22 | 20 | 28 | |
| FODIS | d | 20 | 9 | 0 | 22 | 20 | 28 | |
| CHARCOLOO | | 20 | 9 | 0 | 22 | 0 | 0 | 2340 |
| COMAIRBU | | 20 | 9 | 0 | 22 | 0 | 0 | 9672 |
| L DENSE MD | | 20 | 9 | 0 | 22 | 0 | 0 | 6186 |
| REACHIE MD | | 20 | 9 | 0 | 22 | 0 | 0 | 46 |
| FDR:UE FM | | 20 | 9 | 0 | 22 | 0 | 0 | 170332 |
| FDR:UE FM | | 20 | 9 | 0 | 22 | 0 | 0 | 435644 |
| COFES | d | 20 | 9 | 0 | 22 | 20 | 28 | |
| CESL bas | e | 1980 | 00 | 00 | 00 | 00 | 00 | 212 |
| CESL L2 | e | 1980 | 00 | 00 | 00 | 00 | 00 | 49250 |
| CEST SW | e | 1980 | 00 | 00 | 00 | 00 | 00 | 49680 |
| CEST L2 bak | e | 1980 | 00 | 00 | 00 | 00 | 00 | 49780 |
| BUS88088 AP | | 2005 | 04 | 05 | 18 | 07 | | 50228 |
| Bubble Bobble | | 1982 | 11 | 11 | 11 | 11 | 11 | 11 |

Fig. 19. CA7 EXP output in 85 columns.

Similarly, the output will be even more pleasant at 64 columns:

[illegible]

ՀԱՅԿԻ ԵՄՔ ԳԱՐԿԱՐԻՆԵՐԸ

It's evident that the columns are really 4 and they only got broken down in two lines in order to fit. Let's now examine the use of the `switch` switch. If you use

CAT

Your display now will look similar to this

```

CORES      \      <DIR>
DEMOS      \      <DIR>
DOCS       \      <DIR>
DOT        \      <DIR>
GAMES      \      <DIR>
MACHINES\  \      <DIR>
NEXTZx05.  \      <DIR>
RPI        \      <DIR>
SRC        \      <DIR>
SYS        \      <DIR>
TMP        \      <DIR>
TOOLS      \      <DIR>
LICENSE    .MD      8K
README     .MD      2K
TBBLVE     .FW      168K
TBBLVE     .TBJ     465K

1887M free

0 OK, 0 1

```

As you can see, filenames are now clearly separated at the 3rd character by a dot followed by a 3 letter type. In order to demonstrate what happens with a larger filename we could write a simple program and save it as follows:

```
10 PRINT 'He la huc d'
```

```
58VE This Is A Hello World Program.bas
```

Then try both CAT and CAT as follows

CAT = 'th#.bas'; CAT = th#.bas

Here we're also demonstrating the use of `widow-aligns` (on the first line). Your display will then be


```

THISIS~1.BAS                                1K

1667M free
This Is A Hello World Program.ba
s                                             1K

1667M free

0 OK, 0 1

```

you'll notice that the long filename **This Is A Hello World Program bas** got truncated to its first 8 characters after trimming all space characters followed by a tilde ~ character and the number 1. This is to help differentiate from other files with long filenames that look alike in the first 8 characters of their filename omitting spaces. To demonstrate this type

```
SAVE "This Is A Hello United Kingdom
Program.bas"
```

and

```
SAVE "This Is A.bas"
```

followed by

```
CAT ~*th*.bas
```

The resulting display will now be

```

THISIS~1.BAS                                1K
THISIS~2.BAS                                1K
THISISA~.BAS                                1K

1667M free

0 OK, 0 1

```

As you can see a ~2 was added to the **This Is A Hello United Kingdom Program bas** filename when it was shortened otherwise you couldn't differentiate it from the **This Is A Hello World Program bas** as they both share the same starting characters. As a matter of fact NextZKOS when faced with a lot of similar filenames will keep adding consecutive numbers truncating the original filename further until all the files are displayed in short format. If you now use **CAT** with **EXP** you'll get to see a number of things. First if you don't have a *Real Time Clock* module installed, you will see that all the files you just saved have the same date and time on them and secondly that in the second column, the second character from the left has turned into a from a single dash. This signifies that the archive attribute has been set. **CAT** becomes more powerful with the use of *wildcards* allowing us to get a list of only the files we're interested in omitting all others that may clutter our display. For example

```
CAT ~*.tap'
```

will show us all the *tap format tape image files* we have stored in the current drive and folder.

Thus far we have only displayed the ability to list files contained within the current drive and folder, however **CAT** can display files in different drives, folders, user areas or a combination of the above when the combination is supported by the filesystem of the drive. We can instruct **CAT** to produce listings of files and folders inside drives other than our current drive or folder or even user area without having to change our *default filespec* to that specific area. We'll cover the subject of changing the *default filespec* shortly so for now here are some examples

```
CAT 'm'           Displays a list of all files in drive m
CAT '2m"'         Displays a list of all files in user area 2 of drive m
```


CAI 2m + base

[illegible]

CAT 5 "POTENTIAL"

$$k_{\text{eff}} = \frac{1}{\beta - \frac{\beta - 1}{\alpha}}$$
CAT 6 Performance[®] ++

folder nextxos on drive c

CAT has a lot more options available than CAT which can be seen once you type

• • • 1987

[illegible]

、 5 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039 10

[illegible]

Drive Folder and User Area navigation and management

the storage needs were not as pressing as they are today.

The system is designed to be highly efficient, with a low power consumption of only 100 mW. This is achieved through a combination of hardware and software optimizations. The hardware includes a custom-designed ASIC (Application Specific Integrated Circuit) that handles the majority of the processing. The software is written in C and is optimized for the target hardware. The system is also designed to be highly scalable, allowing it to be used in a wide range of applications.

that followed

rectory tree (it's really an inverted tree with the root of it sitting at the top)

SAVE LOAD

[illegible]

Wasserscheidt, G. 1996. The Wasserscheidt navigation and management program is

MKDIR

MKDIR (for MaKe DiRectory) creates a folder on a drive that supports it. Its syntax is as follows:

MKDIR filespec

[illegible]

If you are using MKDIR with a depth of folders greater than one, the folder name you're specifying should always be a sub-folder of the previous folder. Otherwise you will receive an Invalid path. Below are some examples to illustrate:

| | |
|----------------------------|---|
| MKD R "codes" | Creates a folder named codes under the root drive's root folder. |
| MKD R "codes,codes" | codes is a subfolder named codes under the current drive's root folder inside the codes folder. If there is no folder named codes under the root folder, the command will fail. |
| MKD R "d test" | Creates a folder named test under the d drive's root folder. |
| MKDIR "d test" | Creates a subfolder under the d drive's last changed-to folder. |

The last example is very interesting, as it illustrates how using an array inside of a loop is not a good idea. The number of iterations is not known until the loop is finished, so the array must be created after the loop. This is a common mistake, and it will become very useful when copying as we will see later on.

For `path`, which is the value of `MKDIR` with slashes omitted, and `path` per `mkdir`. As with the fully qualified name of the `MKDIR` version, the `path` may be fully qualified. As with `path`, the `path` may be inside the lifespan the double quotes enclosing it are optional.

RMDIR

AMDIR ~~is a batch file that removes empty folders. It is a free tool that supports finders. Its syntax is as follows:~~

AMDIA fileSpec

While `binlog` is as discussed in `MYKILL` above, `RMDIR` is only available from a local de-
termined disk drive. If the drive is `+`, `RMDIR` is using the default 0 (primary) ex-
ecution engine. The mode is assigned when you would normally all the files and
sub-directories. `RMDIR` always removes the file. `RMDIR` always works with `RMDIR`.
With `+` to work with `RMDIR`, you cannot use `RMDIR *` and expect to remove all
files under the local drive. Any attempt to do so will result in `Bad filename 0 1`
error.

Finally, if you attempt to use AMDiA with a folder that does not exist, you will receive a `InvalidPath` (01 error).

[illegible]

CD

CD (*Change Directory*) changes the current *drive* and/or *folder* (for drives that support folders, or current drive (for drives that do not). **CD**'s syntax is as follows:

CD *filespec*

where *filespec* consists of either one or two of the first two parts of a filename (*Drive* and *Folder*), or filesystems that support folders (FAT 6, FAT32) or of just the *Drive* for filesystems that do not (3DOS, iDPOOS). Setting just the current drive with **CD** is functionally equivalent to using **SAVE**, **LOAD** etc with just the drive as the *filespec*. Unlike folders, there is no way of setting a user area as the default one so if you need to address it, you must do so explicitly through the *filespec*. For example add a **3m** prefix to filenames for files in the user area 3 of drive m. **CD** works with *wildcards* by matching to the first folder in order it finds them and change to that.

CD also accepts three *filespec* shortcuts: (single dot), (double dot) and one of the following: (forward or backward slash). As we mentioned earlier in the chapter, single dot means 'this folder', double dot means 'the folder one level up' and either slash on their own means 'The root folder of the current drive'. Single and double dot entries do not exist on the root folder and therefore you cannot use the shortcuts there.

Using a combination of the double dot and slash shortcuts, **CD** can also easily traverse the folder tree horizontally at the same level without having to write the entire path that precedes the level you're currently in. Obviously, **cd** doesn't make sense at the first level under the root as it would involve much more typing than the slash character alone but it works nonetheless!

Assuming a structure like the one in your **System/Next™** distribution as partly displayed in the figure below, let's provide some examples of horizontal and vertical navigation:



Fig. 2* Folder tree navigation

Let's agree that we're located in the root drive c, and we want to first go to **c:/docs/cpm** and then go to **c:/docs/extra-hw** before returning to **c** again.

We could use one of the following sequences:

```
CD "docs"
CD "cpm"
```

and then

```
CD ".."
CD "extra-hw"
```

and finally


```
CD ".."
CD ".."
```

or alternatively

```
CD "c:/docs/cpm"
CD "c:/docs/extra hw"
CD " "
CD "
```

However it's much less typing to just do

```
CD "/docs/cpm"
CD "/extra hw"
CD "/"
```

It's easy to see that the navigational shortcuts are quicker. The `cd` command equivalent of `CD` is `cd` with the optional switch `verbose` which performs the functions of both `CD` and `PWD` (see below) in order. A small deviation from the syntax of `CD` is that it allows special shortcuts to navigate quickly to the top folder of a deeply nested hierarchy.

These are

| | |
|---------------------|--|
| <code>cd</code> | Functionally equivalent to two successive <code>CD " "</code> commands |
| <code>cd</code> | Functionally equivalent to three successive <code>CD " "</code> commands |
| <code>cd ...</code> | Functionally equivalent to four successive <code>CD " "</code> commands |

PWD

PWD (or Print Working Directory) prints the current drive and folder to the screen or an optional stream number. **PWD**'s syntax is as follows:

PWD [#n]

In a NextZXOS context **PWD** is very useful, however you cannot assign its output to a NextBASIC variable (that easily for use inside our programs. In order to do that, one should be a little creative (skipping ahead to the next chapter) and use the optional *stream* parameter in a manner identical to the trick we used to get time from our R/C pack in Chapter 17. Type

```
DIM d$(255) OPEN #2, 'v>d$' : PWD #2 CLOSE
#2 PRINT d$
```

with which we define a fixed size string variable `d$`, then open stream 2 and assign it to channel `v` which redirects its output to `d$`. We then invoke **PWD** with output redirection to stream 2 which in essence takes its normal screen output and via channel `v` sends it to `d$`, before closing the stream and printing `d$`. We did exactly what **PWD** would do normally (that is print the working directory on the screen) but also managed to store it in a variable for use later.

PWD doesn't have a dot command equivalent with the same name. Instead you only need to use `cd verbose` without a file-spec. The example above therefore becomes

```
DIM d$(255) OPEN #2, 'v>d$' : cd verbose
CLOSE #2 PRINT d$
```

You may notice that there's no stream defined after `cd verbose` and that's because you don't need it as *stream #2* is the screen anyway! It's obvious that the same applies to **PWD** above, but **PWD** does offer the ability to redirect to a *stream* and that illustrated that fact quite nicely. As a matter of fact, you can completely omit the stream from the **PWD** statement in the previous example and it will function in the same manner: you will see why in the next chapter.

Managing files and their attributes

In our examples in this chapter we have managed to clutter our drives with lots of copies of the same programs. This may be desirable at times but sometimes we may want to keep slightly altered versions of the same program in different places (for example to keep a type of version history) but we may not have the organisation of the folders we'll store the files in when we start working.

Other times we may want to get rid of some files we've created for any number of reasons or rename a file from a throwaway name like for example test.bas to something more meaningful and finally we may want to move some files from one place to another when done with them. NextZXOS provides us with all these abilities in the form of the **COPY**, **ERASE** and **MOVE** commands and their dot command equivalents **cp**, **rm** and **mv**.

We'll examine these below and additionally, and how to modify file attributes (what is displayed as the second column in the **CAT EXP** command's output, again via a special version of **MOVE** and its dot command alternative **chmod**). There is no more function provided by NextZXOS in regards to files and that's directly accessing its contents. This however requires the use of *Channels* and *Streams* and is therefore covered in the next chapter.

COPY

COPY does as its name implies. Copies a file from a location to another location. Its syntax is quite simple:

COPY source TO destination

A few notes regarding the differences between source and destination parameters are:

First and most importantly, source can use wildcards while destination cannot. In other words you can write:

```
COPY "c:\*.*bas" TO "m"
```

but you cannot write:

```
COPY "c \*.bas" TO "m \*.bas"
```

or

```
COPY "c \*.bas" TO "m \a*.bas"
```

as any attempt to do so will generate a **Destination cannot be wild 0:1 err #**

Secondly, copying files between filesystems with different capabilities will perform some form of translation to the filenames. To give an example with two files named **raycaster** (as longer than 15 characters) and **later.bas** starting with two dots on drive c: doing:

```
COPY "c \*.bas" TO "m"
```

will change the filenames to **raycas~1.bas** and **later.bas** as the *RAMdisk* is a *3DOS* drive and as such accepts only 8+3 filenames.

Thirdly, the destination is not checked for if the files being copied already exist. So, you perform the above operation twice, each time **COPY** will replace the files on the destination without creating backup files except if the file named the same in the destination has the *protected* attribute set. To demonstrate let's skip a bit ahead and introduce you to an attribute setting command. Type the following:


```
COPY "c:/nextzxos/pisid.%" TO "m
MOVE "m PISID.BAS" TO "+p"
COPY "c:/nextzxos/pisid.%" TO "m
```

The first **COPY** operation will succeed while the second **COPY** operation will fail in the case of a mass **COPY** if the operation fails for any file it will fail for all remaining files so keep that in mind.

COPY does not work between a disk and a tape doing for example

```
tapeout test.tap"
COPY "m *.bas" TO "t"
```

will fail with a Destination must be path, 0:1 error. Note above the use of the **tapeout** dot command which we will cover later on. It just allows us to substitute a tape image file for an actual tape. To perform the above function we will need to do the following

```
.tapeout "test.tap"
LOAD "m hello.bas"
SAVE "t hello.bas"
```

and verify the output with **lstap** we covered earlier

```
lstap "hello.tap"
```

or alternatively not use **tapeout** and **lstap** at all and save onto an actual tape (in which case we'd use **VERIFY** to check if the file was actually written)

There is a special version of **COPY** where the source file is stripped of all control codes (i.e. maintaining *End-Of-line* characters 'CR', 'LF' or the combination of both). See Appendix A for all Control Codes. It exists as either shortcuts **SCREEN\$** and **LPRINT** in lieu of destination (or as any stream that can be attached to a channel). The **SCREEN\$** shortcut gets any file and prints it on screen while the **LPRINT** shortcut gets any file and sends it to a ZX Printer or compatible. A good way to test the functionality is to check some of the documents in *zxdos*. For example to see the pinouts of the Next board you can type

```
COPY "c:/docs/extra/hw/pinouts/pine.txt"
TO SCREEN$
```

while if you do

```
COPY "c:/docs/extra/hw/pinouts/pine.txt"
TO LPRINT
```

the file will be sent straight to the printer. **SCREEN\$** and **LPRINT** are shortcuts for their respective streams (as you will see in the next chapter). Although there are no shortcut keywords for other streams, if the destination is set to any stream **COPY**'s behaviour will be identical to what we just saw.

The **dot** command equivalent for **COPY** is **cp** and its syntax is similar with the exception of the **force** switch which allows overwriting of files without prompt. **cp** is **ANNO** currently address + **SDOS/GEIOS** drives so it should be only used on **FAT** partitions on the SD Card.

ERASE

Files can be deleted from a drive using the **ERASE** command. Its syntax is as simple as one would imagine

ERASE filespec

where **filespec** follows the same conventions as **CAT** meaning that just like **CAT** you can use the wildcards ***** and **?** to identify a group of files (i.e. you can specify the filename if full (including optional Drive and/or User Area and Path) or you only want to get rid of one par-

icular file. **ERASE** offers you some form of protection if your *filespec* contains *wildcards* in the form of a question in which you will have to answer with a Y on the keyboard to continue or with N to stop, but offers no protection if you specify a single filename which will immediately be erased from the drive – so extend *caution*! For example, you wanted to delete a file from drive m, called FRED.BAS, you would use

```
ERASE 'm fred.bas'
```

If drive m has already been set as the default drive (by either using **SAVE LOAD** or even **CD**), then you don't need to introduce the m at the start of the filename. It doesn't hurt to include the drive anyway, and with as powerful a command as **ERASE** is, you might feel safer if you do. To erase all the files on drive d, you would use

```
ERASE 'd * *
```

Before doing this, NextZXOS will ask for confirmation by printing

```
Erase d * * ? (Y/N)
```

on the bottom of the screen and assuming that you really mean to wipe all the files from the disk in drive d, you would then type Y.

If you attempt to delete a single file, or a group of files, using *wildcards*, while there are no files on the drive that match the *filespec*, a **File not found** error will be displayed.

The dot command equivalent to **ERASE** is called **rm** (from remove) and its syntax follows that of **ERASE** with the exception of two switches namely **verbose** and **help**.

MOVE

MOVE is a very powerful command. It performs a total of five functions: moving and renaming files, changing file attributes and manually mounting and dismounting drives. Since there are separate sections for the last three functions, we'll cover only the first two here. For moving and renaming, **MOVE**'s syntax is

```
MOVE source filespec TO destination filespec
```

where *source filespec* and *destination filespec* follow everything discussed in the *File names* section earlier with the following considerations:

- You cannot use *wildcards* in either the source or the destination. This means that both source and destination have to be complete filenames.
- You cannot perform a **MOVE** operation between drives.

Let's examine what will happen in the first case. Assuming you have 3 Next BASIC files named HELLO1.BAS, HELLO2.BAS and HELLO3.BAS in drive m, in the default User Area 0) and you want to move them to User Area 1 (typing as you would probably expect:

```
MOVE "*.bas" TO "1 "
```

will fail with **Bad Filename, 0:1**. To perform this you should actually do

```
COPY "*.bas" TO "1 "
```

followed by

```
ERASE '* bas'
```

in the second case, and since we now learned our lesson we won't be using wildcards, attempting to **MOVE** one file between drives like so

```
MOVE "c \test.bas" TO "d \test.bas"
```

will fail with **No rename between drives 0:1**. To perform this you should actually do like above


```
COPY "c:/test.bas" TO "d/"
ERASE "c:/test.bas"
```

As you probably have already figured out, moving and renaming files is basically the same procedure and since we have to write an entire filename in both source and destination we can change it at the same time:

```
MOVE "hello1.bas" TO "c:/bak/hello.bak"
```

both moves locations and renames hello1.bas

Imagine we have saved a file called **FRED** and then after working on it and saving a new version with the same name, realised that we had made a terrible mistake and would like to recover the last version. This would be possible using the commands:

```
ERASE *fred
MOVE "fred.bak" TO "fred"
```

If a file you're moving or renaming already exists (or rather another file with the same name at the intended destination) **MOVE** will fail with an **Already exists 0:1** error.

MOVE's dot command alternative is **mv** and, unlike other dot command alternatives we've examined so far, its renaming and moving capabilities far exceed those of **MOVE**'s. It allows operations across different drives, *interactive* or *automatic* overwriting of already existing files as well as the full use of *wildcards*. Its syntax is:

```
mv [OPTION] [T] source destination -or-
mv [OPTION] source DIR -or-
mv [OPTION] -i DIR source
```

Where *source* and *destination* can be any valid NextZXOS filespec, including *wildcards* and *DIR* is any valid folder. *Source* or *Destination* filespecs with trailing slash characters or *.* are considered to be folders. As **mv** has numerous options, they are listed in the table below to help you better understand what it can do. In general, when you have a large quantity of files to be moved or renamed it's better to use **mv** over **MOVE**.

| Option | Alt Option Syntax | Description | Notes |
|--------|-------------------------|--|---|
| -b | | Makes backup of existing destination | |
| | -force | Do not prompt for overwrite | Of these three options, the last one (-f) is the one that takes effect. |
| | -interactive | Prompt for overwrite | |
| -f | -no-clobber | Do not overwrite | |
| | -strip-trailing-slashes | Remove slashes from names | |
| -S | -suffix Suffix | Override default backup suffix with Suffix | |
| | -system | Match system files in source | |
| DIR | -target directory=DIR | Move everything in source to folder DIR | |
| -t | -no-target-directory | Treat destination as a normal file | |
| -u | -update | Move only if source is newer than destination or destination doesn't exist | |
| -v | -verbose | Explain what is being done | |
| -h | -help | Prints it's list of options | |
| -V | -version | Prints the version of mv and exits | |

Table 16 mv options

File attributes

As mentioned in the previous section, **MOVE** has another use besides renaming and moving files and that is to change a file's attributes. Attributes are bits of information associated with a file that tell you (and the computer) a little more about it. You already saw in the **CAT EXP** and **ERASE** examples how attributes appear to you and how they can affect your files. There are three attributes that can be changed plus one more that is automatically managed: write protection, system status and archive. The most useful attribute is as

we've seen already: *write protection*. Once a file's write protection attribute has been set, it will not be possible to erase it, or save a file with the same name, until you remove it.

MOVE's syntax for attribute changing is a bit different from the one used for renaming/moving.

MOVE *filespec* **TO** *+/attribute*

Where *filespec* CAN include *wildcards* unlike the previous case, and *attribute* is one of the following letters: **p**, **a** and **s**, used with either a **+** or **-** prefix. The prefix serves as a set (or and unset/clear) for: **p**'s short for **protection**, **a**'s short for **archive** and **s** is short for **system**.

Write protection is the most useful attribute for NextZXOS. Try

```
MOVE "hello.bas" to "+p"
```

If you now try

```
ERASE "hello.bas"
```

ERASE will fail with a File is read only error.

To switch write protection off type

```
MOVE "hello.bas" TO "-p"
```

and you'll be able to erase the file as before.

As mentioned, we can use wildcards when changing attributes. As an example, to make all the files on drive **m:** write protected, you would type

```
MOVE "m.*.*" to "+p"
```

As always, the drive letter can be omitted if it is the current default drive.

You can repeatedly switch attributes on or off without causing an error, so if you set write protect on a file that has already got write protection, it will just stay protected.

The second attribute we mentioned is the *system status attribute*. This is really provided just to be compatible with other CP/M-based computers, however, if you do set a file's *system attribute* to **off**, you will see that the file no longer appears in the list when doing a normal **CAT**. It will appear however when using **CAT EXP** with an **s** marked in the second column and when using **ls**. Try the following:

```
MOVE "hello.bas" TO "+s"
CAT
CAT EXP
LOAD "hello.bas" RUN
```

As you can see **hello.bas** became invisible to **CAT** but you can still **LOAD** it properly if you know its name. Bear in mind that you cannot have two files on the same disk with the same filename and different system status attributes, so if you try to create or copy a file onto a disk where a file of that already exists (but is hidden from **CAT**), then the previous file will be deleted, unless of course its *write protect attribute* is set.

The final attribute you can change is known as the *archive attribute*. In an expanded catalogue, it shows up as a **A**. In other systems the *archive bit* is cleared when a copy operation has been performed, but that doesn't happen on NextZXOS. NextZXOS automatically sets the *archive bit* when saving on a FAT driver but doesn't do so on DE DOS or 3DOS drives, so is therefore of no practical use and is only provided for file compatibility with CP/M.

If you try to use any letter other than **a**, **s** or **p** in setting or removing attributes, or if the attribute option string is not two characters long, then you will receive an Invalid attribute error.

As we saw in Chapter 18, the `EXP` command is used to export data from a virtual device to a physical device. In this case, we will use the `EXP` command to export data from the RAMdisk to a physical device. The syntax for the `EXP` command is as follows:

```
EXP [device] [path] [file]
```

where `device` is the name of the physical device, `path` is the path to the file, and `file` is the name of the file to be exported.

```
CHMOD TBSUP PD H
```

you will see that nothing has changed when doing `CAT EXP`. If you however take your SD Card to a PC, you will be able to see the file again there.

The RAMdisk

You may have seen a warning when you started NextZXS that there is a warning about the RAMdisk. This is because the RAMdisk is a virtual device and it is not possible to write to it directly. However, you can use the `MERGE` command to merge data from the RAMdisk to a physical device. The syntax for the `MERGE` command is as follows:

```
MERGE [filename] [path] [file]
```

where `filename` is the name of the file to be merged, `path` is the path to the file, and `file` is the name of the file to be merged to.

As we saw in Chapter 17, one of the more interesting uses of the RAMdisk is as a temporary storage area. This is because the RAMdisk is a virtual device and it is not possible to write to it directly. However, you can use the `BANK` command to bank data to the RAMdisk. The syntax for the `BANK` command is as follows:

```
BANK [device] [path] [file]
```

where `device` is the name of the physical device, `path` is the path to the file, and `file` is the name of the file to be banked.

Drive and Partition Management

We've talked about physical drives and virtual devices. We've also talked about the different file systems that can be used. In this section, we will talk about how to manage drives and partitions. The first command we will talk about is `CATTAB`. The syntax for the `CATTAB` command is as follows:

```
CATTAB [device] [path] [file]
```

where `device` is the name of the physical device, `path` is the path to the file, and `file` is the name of the file to be created.

The second command we will talk about is `CATASN`. The syntax for the `CATASN` command is as follows:

```
CATASN [device] [path] [file]
```

where `device` is the name of the physical device, `path` is the path to the file, and `file` is the name of the file to be created.

The third command we will talk about is `MOVE`. The syntax for the `MOVE` command is as follows:

```
MOVE [device] [path] [file]
```

where `device` is the name of the physical device, `path` is the path to the file, and `file` is the name of the file to be moved.

The fourth command we will talk about is `MOVE OUT`. The syntax for the `MOVE OUT` command is as follows:

```
MOVE OUT [device] [path] [file]
```

where `device` is the name of the physical device, `path` is the path to the file, and `file` is the name of the file to be moved out.

The fifth command we will talk about is `RMOUNT`. The syntax for the `RMOUNT` command is as follows:

```
RMOUNT [device] [path] [file]
```

where `device` is the name of the physical device, `path` is the path to the file, and `file` is the name of the file to be mounted.

Next, we will talk about the `mkdata` and `mkswap` commands. The syntax for the `mkdata` command is as follows:

```
mkdata [device] [path] [file]
```

where `device` is the name of the physical device, `path` is the path to the file, and `file` is the name of the file to be created.

The syntax for the `mkswap` command is as follows:

```
mkswap [device] [path] [file]
```

where `device` is the name of the physical device, `path` is the path to the file, and `file` is the name of the file to be created.

CAT TAB and CAT ASN

CATTAB lists the storage devices currently connected to your ZX Spectrum Next and their partitions. It's syntax is

```
CAT [#n] TAB
```

where `#n` is the number of the device. For example, `CAT 1 TAB` will list the first device connected to your ZX Spectrum Next with a single SD Card reader, giving

```
CAT TAB
```

will return

```
MHC Unit 0 (1024M)
MHC Unit 5 (1024M)
5 1 NEXT          1024M FAT32
```

which illustrates also a point we made early in the chapter. Even if a reader is assigned two drives, only the first is used. In this case, the first drive is used for the `MHC Unit 0` and the second drive is used for the `MHC Unit 5`. If we had a single SD Card reader, the display would have been


```

MMC Unit 0 (1024M)
0>PLUS1DEDOS          64k sys
0>General              4096k data
0>CPM A                120k data
0>CPM B                512k data
0>CPMstuff             512k data
0>Dev                  256k data
0>Next                 320k data
0>"*****"           10304k FREE
24 free partition entries
MMC Unit 5 (1024M)
5>1>NEXT              1008M FAT32

```

CAT ASN on the other hand, displays which partition of disk is assigned to which drive. The syntax is similar to CAT TAB

CAT [#n] ASN

where again #n is an optional stream for the output to be redirected to. On a standard ZX Spectrum Next with a single SD reader and prepared CPM (whose virtual drive a, as we have discussed would be already automounted) giving

```
CAT ASN
```

would produce the following output

```

A      Mounted F5
C 5>1>NEXT
M 4>RAMdisk

```

— you are asking what happened to the *DEDOS* partition we displayed earlier. It's not mounted because *DEDOS* partitions do not auto-mount. To mount them, or any other partition of virtual/physical disk, you will need to employ the following commands

MOUNT IN MOVE OUT and REMOUNT

In order to assign (mount) a disk/partition or virtual/physical disk to a drive you need **MOUNT**. Its syntax is as follows

MOUNT drive N mount point

where drive is any valid NextZXOS drive 'a' to 'p' and mount point is either a device >{partition} or >{partition name} or a file-spec of a virtual disk. Devices that don't have partitions are written as X> where X is the device number, while devices that have partitions are written as X>Y>{partition name} where Y is the partition number for FAT partitions and X>{partition name} for *DEDOS* partitions. In the case of *DEDOS* partitions the number can be initially omitted as well if on device 0. Assuming that we had unmounted the RAMdisk, in order to mount it again in some other drive, we'd need to do

```
MOUNT "0" IN "4>"
```

Notice that there's no partition number following the 4> as the RAMdisk has no partitions. To mount a +3 disk image named *mike.dsk* located in *c:/images*, into drive *b* we would need to

```
MOUNT "b" IN "c:/images/mike.dsk"
```

Whereas to mount an *DEDOS* partition (for example one of the ones we examined earlier) you would have to

```
MOUNT "e" IN "0>CPMSTUFF"
```

or


```
MOVE "a" IN "/CPMstuff"
```

Attempting to mount a drive that's already assigned will produce the error **Already exists**. If in order to do that, you first need to unmount the drive with **MOVE OUT**. The syntax is even simpler:

```
MOVE drive OUT
```

So to unmount the disk image from **b**, we just need to give

```
MOVE "b" OUT
```

You cannot unmount the **c** drive and attempting to do so will report an **in use** or **01** error. You can however temporarily eject it (for example to write to it or just change it to a different version of NextZXOS or even a game). Doing ~~that way~~ powering down or just arbitrarily can damage your card beyond repair so you must be **VERY** careful. Since the potential for damage is great, NextZXOS has a special command to address that specific need called **RE MOUNT**. Remount is given without any parameters and upon invocation it will prompt you to

```
Remove/insert SD and press Y
```

Once you see the message you can eject your SD card and when you reinsert it, press **Y**. NextZXOS will perform the same mounting procedure it performs at boot for all drives and your SD card contents will be safe.

Virtual filesystem management `mkdata` and `mkswap`

As we've already demonstrated, NextZXOS can read unprotected FATDOS and IDEDOS virtual disks, but how are these made? There are two ways to do it: We can either create them externally using special imaging software or right in NextZXOS, with the use of a specialised **dot** command called **mkdata**. Its syntax is as follows:

```
.mkdata filespec [size]
```

where *filespec* must follow the requirements set forth in the *Filenames* section for *loga* filenames, omitting the drive and size is an optional number from 1 to 16 in Megabytes. Leaving size blank will select the default size of 8 Megabytes. You can use ANY filename, however, only filenames with a **.p3d** type, named as described in the automounting section earlier in this chapter and located inside **c:/nextzxos**, will be automounted. Here are some examples:

To make an 8 Megabyte automountable (as a) virtual disk

```
mkdata /nextzxos drv a p3d 8
```

To make a 8 Megabyte virtual disk that can be manually mounted in **c:/images**,

```
mkdata /images/disk p3d
```

In order to make a virtual disk in a different drive you need to first change to it. For example

```
CD "d"
mkdata /images/disk p3d
```

will make a 8 Megabyte virtual disk image file named **disk.p3d** in **d:/images**.

NextZXOS also supports virtual memory in the form of virtual swap partitions. These are similar to the virtual disk images with the difference that they cannot be mounted as drives. You can make virtual swap partition images with the **mkswap dot** command which follows the same syntax as **mkdata**:

```
.mkswap filespec [size]
```

To make an 8 Megabyte virtual swap partition image named **swp.1.p3s** you will need to give


```
.mkswap /nextzxos/swp 0.p3s 8
```

Swap partitions named `swp-0 p3s` to `swp-9 p3s` which are present in the `/nextzxos/` folder will be available for machine-code application programs to use (via the *IDEOS* API).

Printing

NextZXOS supports printing via ZX Printer, Timex Sinclair 2040 and compatibles like the Alpha model. It also supports printing via the *Win* module if there is installed and you have access to a "tpsta" printer or a printer compatible with D. Rimmer's *PrintShop* as found on <https://github.com/StalePixels/PrintShop>.

To print a string you only need the `LIST` command while to print any string to the printer you need to use `PRINT` (yes, and you'd screens can also be printed by using the `COPY` command given by itself with no options. In order to demonstrate this we will have to jump a bit ahead: load one of the games from `c/games/Classic40`, preferably into windowed screen. Once you see the screen press the `NMI` button (the left side of your ZX Spectrum Next A menu will appear, using the cursor keys go to the *Screenshot* menu and press `ENTER`. Click *Standard* and Press `ENTER`. Press `SPACE` and type in a name (for example `test.scr`) press `ENTER` again and then press the reset button on the side of your computer or `F4` of your keyboard. Re-enter NextBASIC and navigate to the location you were in. Then do the following:

```
LOAD "test.scr" SCREEN$ COPY
```

The screenshot will print on your printer.

Since you're undoubtedly observant you may have seen the *Print* form in the *Screenshot* sub-menu when you pressed the `NMI` button. That will do the exact same thing. But more on that in a later section below. There are also other ways to print which we will examine in the next chapter.

The SPECTRUM command

`SPECTRUM` is a command that's a bit of a hack. It takes all the switches modes key program in various snapshot formats, change colour schemes, adjust the displayed columns of the editor and finally format and adjust the screen save function. Let's start with the simplest iteration of `SPECTRUM` which is the command without any options. This will take us into 48K mode preserving any NextBASIC program we have in memory but using all vector fonts (except for the dot fonts) and while this will be still available the program you have loaded in memory is using specialised NextBASIC features. `LIST` may produce garblish-like graphics in the place of where commands would have been, and running it will probably produce a *C Nonsense in BASIC* error (let's demons, we'll type

```
LOAD "c:/nextzxos/mo/inter.bas"
LIST
SPECTRUM
LIST
RUN
```

If you are in the standard ZX 48K mode you will need to know the keywords printed on your keyboard. Assuming you can find where `CAT` is press `EXTEND` (or `SYMBOL`, `SHIFT` and `0`) type

```
CAT
```

you will receive an *Invalid stream 0 1* error. That's because 48K ZX Basic is unaware of any class of byte medium except for the ZX Microdrive and `CAT` is made to work with

6. If the screen is a picture, it will be in your display. The display is damaged then (yes, they are displaying the screen). If it is a picture, it will be in your display. The display is damaged then (yes, they are displaying the screen). If it is a picture, it will be in your display. The display is damaged then (yes, they are displaying the screen).

that in order to actually see what's on your drive, you will need the *dir* command equivalent of **CAT**.⁷ Indeed typing

```
^ \ S
```

you will once again, see what's on your drive.

Once **SPECTRUM** is used to change to 48K Mode, you cannot return to the Next mode using a command (as **SPECTRUM** does not exist in 48K BASIC). Instead you will have to 'reset' your machine, using either the Reset button on the side of the computer or by pressing **NMI** together with **1**.

A more complex iteration of the command is the following:

SPECTRUM filespec

This command loads a snapshot file in the popular **z80**, **ena**, **snx**, **p** and **o** formats and runs it. 48K, 128k, as well as ZX80 and ZX81 snapshots are supported. Here are some examples:

To load the ZX81 classic 3D Monster Maze

```
SPECTRUM "/games/zx81/3dmm/3dmonstermaze.p"
```

To load Pogie in Dreamworld Demo

```
SPECTRUM "/games/next/pogie/pogie.snx"
```

To load Darkstar

```
SPECTRUM "/games/classic128/  
darkstar.z80"
```

Two more specific variations of the basic **SPECTRUM** command are:

SPECTRUM LOAD

which switches to the ZX Spectrum 128k compatibility mode and invokes the Tape Loader and

SPECTRUM 48

which switches to the ZX Spectrum 48k compatibility mode, without any access to 128k hardware features. Both of those are mostly of use to the **TAP**, **TZX**, Tape Loaders included with *NextZXOS*.

To change colour schemes for the *NextBASIC Editor*, **SPECTRUM** can be used with one of the following modifiers: **INK**, **PAPER**, **FLASH**, **BRIGHT** and **ATTR** (which sets all the previous ones in one command). The syntax is as follows:

SPECTRUM MODIFIER n

where **MODIFIER** is one of **INK**, **PAPER**, **FLASH**, **BRIGHT** or **ATTR** and **n** is a standard colour from 0...7 when using the **INK** and **PAPER** modifiers, 0 or 1 for *disabled* or *enabled* when using the **BRIGHT** and **FLASH** modifiers, or calculated as *128*flash*, *164*bright*, *18*paper* + *ink* for the **ATTR** modifier. Here are some examples:

```
SPECTRUM INK 4 SPECTRUM PAPER 0
```

or

```
SPECTRUM ATTR 4
```

⁷ The *dir* type is essentially the same as the basic **SPECTRUM** command, but snapshots using some fixed mode settings, such as example ZX81 DMA instead of ZX81 DMA, and a previous features (never supported).

both set the *NextBASIC Editor* colours to green ink on black paper. You can see how the second line is derived by doing the following calculation: $(128*0) + (64*0) + (8*0) + 4$

SPECTRUM PAPER 1 SPECTRUM INK 6

or

SPECTRUM ATTR 14

set the *NextBASIC Editor* colours to yellow ink on blue paper. Try to figure out how the second variation works.

The colour scheme applies to the standard 32 column editing mode as well as the hi-resolution 64/85 column modes. However, since *ayer 1.2* only allows 8 different colour schemes, the scheme used is the one with the same **PAPER** colour as standard mode.

SPECTRUM can also be used with the **CHR\$** modifier to set the number of columns in the *NextBASIC* editor. Its syntax is

SPECTRUM CHR\$ n

where *n* is one of 32, 64 or 85 for the available column modes. To switch for example to 64 column mode you should type

SPECTRUM CHR\$ 64

Attempting to enter a value other than 32, 64 or 85 as parameter will produce an **Integer out of range 0:1 error**.

Finally, **SPECTRUM** used with the modifier **SCREEN\$** can control the *NextZXOS* screensaver behaviour. The syntax is as follows:

SPECTRUM SCREEN\$ n,f

where *n* is the type of screensaver: 0 = bouncing box, 1 = blank screen, and *f* is the timeout in minutes from 0 to 127. If *f* is 0 then the screensaver is *disabled* until the next reset. The screensaver will activate after the selected timeout whenever the machine is waiting for a key to be pressed under the following circumstances:

- In menus: *Browser*, *Calculator*, *NextBASIC Editor* or while in the *Command Line*
- During **INPUT** statements
- During **PAUSE 0** statements
- When **NEXT #n, var** is waiting for a keystroke from the **K \$** or **W channels**
- When executing machine-code software that uses the **IDF \$BROWS+R** call or the **IDF \$STREAM** IN call, accessing **K \$** or **W channels** or an **IDE BASIC** call accessing the previously listed *NextBASIC* statements

The screensaver will not activate when games are being run (unless they use the AP calls listed above) or in 48 BASIC.

Speed Control

The *ZX Spectrum Next* has a much faster CPU than its predecessors operating in one of the following speeds: 3.5MHz (same as the original *ZX Spectrum*), 7MHz, 14MHz and finally 28MHz. *NextBASIC* by default will set the CPU to execute at 3.5MHz, a setting which can be changed using either the left and right cursor keys while in any *NextZXOS* menu or directly from *NextBASIC* by using the **RUN AT** command. The syntax of the latter is as follows:

RUN AT s

where *s* is a number from 0 to 3 (0 = 3.5 MHz, 1 = 7 MHz, 2 = 14 MHz and 3 = 28 MHz). For example, to execute a program at 28 MHz begin the program with a

1 RUN AT 3

NextBASIC Editor and Program support commands

NextZXOS provides a few direct commands that allow NextBASIC programmers to control both the appearance as well as the flow of their programs. These are:

ERASE [*first last*]

erases all lines between *first* and *last*, inclusive, keeping any variable intact. **ERASE** or its own deletes the entire program, still keeping all variables intact, and unlike its parameter version, can be included in a program, see the *autoexec.bas* section below for an example.

LINE *first step*

renumbers the program starting at line *first*, using a predefined *step*. Let's assume a small program:

```
10 FOR f=1 TO 10
20 PRINT f,
30 NEXT f
```

we now give

```
LINE 2 3
```

The program becomes:

```
2 FOR f=1 TO 10
5 PRINT f
6 NEXT f
```

It's obvious that we can pack as much program as we can in the amount of lines NextBASIC allows once our program is finalised. This should not be confused with the direct command:

BANK n LINE *first last*

which copies all lines in the main program between *first* and *last* to **BANK** number *n*. More on all bank-related commands can be found on *Chapter 23*.

LINE MERGE *first last*

performs an even nicer optimisation to our typed programs, merging lines together to form a longer line, thus freeing lines for use. Assuming the program above, type:

```
LINE MERGE 2,6
```

the program then becomes:

```
2 FOR f=1 TO 10 PRINT f,
NEXT f
```

Obviously **LINE MERGE** makes our programs less readable but let's us pack them even more allowing for even more line numbers to be freed.

BANK n MERGE

copies a banked program back into the main program (more details on *Chapter 23*) erasing everything that's already there with the same line numbers. For example, in the above **LINE MERGE** example, **EDIT** line 2 to be also line 4 by going over line number 2, deleting it and replacing it with a 4. Then do the following:


```

BANK NEW 3
BANK 3 LINE 2,2
ERASE 2,2
LIST

```

and finally

```
BANK 3 MERGE
```

You'll see that the line you erased with **ERASE 2,2**, is back into place.

NextZXOS also provides one more command we've already seen but haven't sufficiently explained yet.

```
BANK n LIST #r [PROC name()]
```

which just lists the program, and optionally redirects its output to a stream that's currently in memory. Optionally **LIST** can produce the list of the program that's currently in **BANK n** or list a program whether banked or not starting with the procedure **name()**.

%CODE

NextBASIC options are controlled by the special **%CODE** integer variable. This is reset to zero when a program is loaded/run.

Currently available options are

| Bit | Use |
|-----|--|
| 0 | if set, %AND n and AND(n) return values between 0..n, rather than 0..1 |
| 1 | if set, the BREAK key is disabled |

The Browser

In order to allow easier navigation of your files, NextZXOS comes with the *Browser*, a program that allows you to do so in a visual way. The *Browser* features the following:

- Easy navigation of drives and folders
- File management facilities: copying, erasing, renaming and moving of files
- Quick virtual disk mapping
- Automatic launching of known file types
- Extensible architecture for launching
- Cursor key or joystick navigation

The *Browser* is launched by using the **EDIT** key to bring up the *NextZXOS* menus or directly upon bootup by selecting the first entry in the *NextZXOS* Startup menu.

The Browser Window

Once the menu is selected and **ENTER** is pressed, the screen changes to the *Browser* window containing a list of the files located in the *default drive and folder*, as set by the **CD**, **LOAD**, **SAVE**, **MERGE** or **VERIFY** commands in NextBASIC. Normally, upon initial boot, this will be **C:**, but subsequent runs without a complete power down may show different locations reflecting the last drive and folder set as default. Note that you do not need to switch to NextBASIC to set a default drive and folder. Whatever you select with the *Browser* has the exact same effect for NextBASIC as giving one of the aforementioned commands.

The Browser window consists of five separate areas, as seen in the figure below:

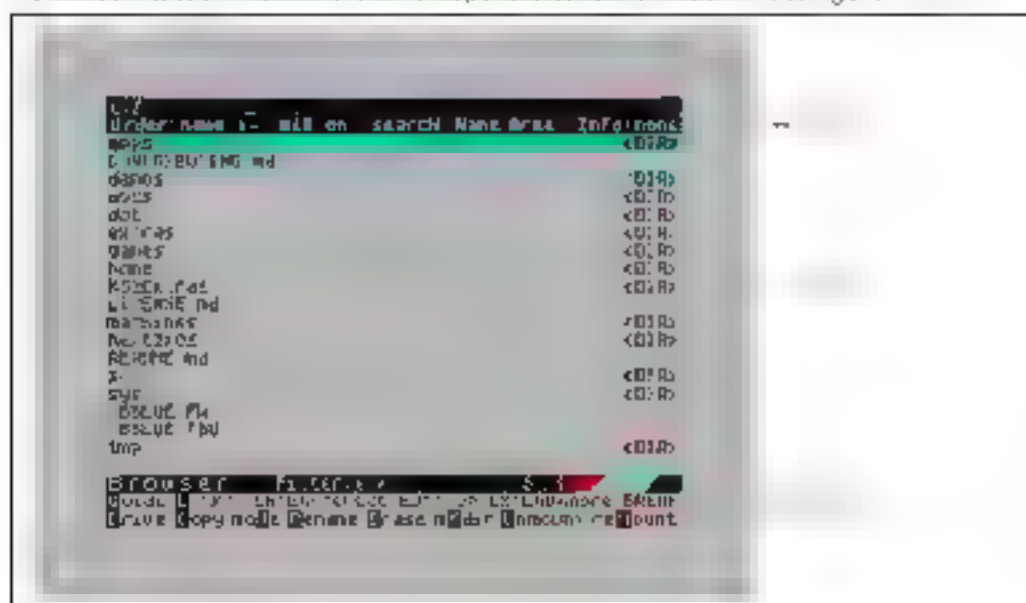


Fig. 22 Browser window areas and their function

On the top of the Browser window is the **Current Drive and Path Area**. As you navigate your drives, it changes to reflect the current drive and folder you're in. This in effect is the same as giving the **PWD** command when in **NextBASIC**.

Right below that are the **View Options** which control the way the list will appear from what info it will show to how the list is sorted together with a search control.

Immediately after is the **File and Folder List Area**. It contains all files and folders at the point you're located as reflected by the **Current Drive and Path Area** at the top in combination with the **Active File Filter Area** that's right below it. (more about that in a little bit) It shows in pages of 19 items at a time. You navigate the file and folder list with the cursor keys **ENTER** and **EDIT** (a joystick set as cursor, or the first Kempston or Mega drive joystick regardless of what port (Left or Right) it's set to). Immediately below the **File and Folder List Area** is the **Active File Filter Area** with which, you can reduce the file list to whatever types (including folders which have essentially a blank type) you wish to see according to a filter set by wildcards and finally the bottom two lines is the **Info/Status and Commands Area**.

Using the Browser

The Browser is extremely easy to use, all it takes is a few keystrokes to accomplish most tasks. Controls are listed in the next table:

| Key | Description |
|----------------------|--|
| ↑ | Moves one page up or to the topmost item if you're on the first page |
| ↓ | Moves one page down or to the last item if you are on the last page |
| ← | Move up one item |
| → | Move down one item |
| ENTER | If it's a folder, change to that folder. If it's a file, attempt to execute it |
| SYMBOL SHIFT + ENTER | Attempt the secondary action stored in browser.cfg for the file type |
| EDIT | Move up one folder |

Table 7 Browser controls

while commands are the following:

| Key | Description |
|-----|---|
| Q | Cyclically changes the sorting of the files between Name, Size and Date |
| + | Sorts the display incrementally |
| - | Sorts the display Decrementally |
| X | Toggles whether Folders and Files will be mixed or separate |
| W | Performs a search for a specific file |
| N | Shows the full name of the currently selected object |
| A | Switches Jsa: Area |
| I | Toggles the info display between Name, Size, Date and Attributes |
| D | Cyclically changes the drive to the next in the list of mounted drives |
| K | Makes a new folder |
| R | Renames the currently selected item |
| C | Selects the currently highlighted file for copying |
| E | Erases the currently selected item |
| M | Mounts current drive |
| V | Unmount current drive |

1. **Copy** - will copy the selected text to the clipboard. **Copy?** Y/N (if you press Y, the text will be copied to the clipboard. If you press N, the text will not be copied to the clipboard. If you press any other key, the text will be copied to the clipboard.)

Erase also asks a similar question. Erase? (Y/N) will appear after you highlight a file and press **Enter**. The answer should be **Y** or **N**. **Dir Full** is only displayed if you have selected the **Full** option. If the folder contains any item in it.

When using the **MOVE** command, we can't if we only delete the things we want to move. Let's try to get an error. Run a **New** name command on the **old** and then give it a new name. Now if we give it the **old** name back, we'll get an error. Here it does it with a few tries. Finally, it's deleted the old name, which means that it's all of the new name was left. This is the error. As you can see, the error only gives the file name and the beginning of the new name. It will only give the name between drives error.

Y is at the same level as a folder, the folder is the folder, however the path is 3
ready exist otherwise Rename will fail with an **Invalid path error**

To make a new identifier by the keyword `as` please use the command `as result` by passing `K` as your keyword. Then `as` and `Area` will have the same display name. The new identifier `as` is not implemented in the MKDIR command yet. As is the case with MKDIR any attempt to pass `as` as a drive letter or a switch option will result in `Not implemented` error. `as` and `Area`

[illegible]

There is always an enormous role played by the land if you do have that you are
imposed through the process when you have been involved in a particular sample of
the land and you have to be very clear about the way that you will be able to
do it in a way that is acceptable.

remove your SID card, this message applies to both SID cards, and once you press Y on the prompt, NextZXOS will perform the REMOUNT command as discussed earlier, thus remounting any physical drives you've unmounted.

Configuring the Browser

File and drive management operations with the Browser is one facet of what it can do. The most important function it has however, is to recognise and launch files of various types when we highlight them and press ENTER (or SYMBOL SHIFT + ENTER – see immediately below). It's able to do so due to its extensible nature using a simple, specially formatted text file called `browser.cfg` that's located under `c:\nextzxos\`. The Browser also offers a way to assign TWO types of launching for a filetype. This is accomplished by adding two lines in `browser.cfg`. For example we could LOAD a `bas` file, it could be plain text, using the `bas2txt dot command`. The first action would be launched by ENTER and the second one with SYMBOL SHIFT + ENTER.

Each line of `browser.cfg` contains information formed in the following fashion:

`TYPE LINE`

where TYPE is a 3 letter file type, e.g. `BAS`, followed by LINE which is a sequence of NextBASIC commands separated by colon characters as per usual, but prefixed with one of the following symbols:

| Prefix | Meaning |
|-------------------|--------------------------------|
| <code><</code> | Return to Menu afterwards |
| <code>></code> | Return to Browser afterwards |
| <code>~</code> | Return to NextBASIC afterwards |

The NextBASIC commands that follow use the following placeholders:

| Character | Meaning |
|-----------------|---|
| <code>%</code> | s replaced by the short filename as read by the Browser ¹⁸ |
| <code>%"</code> | s replaced by the long filename as read by the Browser and must be terminated by a matching quote (") |
| <code>%</code> | s replaced by language code file: en for English, es for Spanish etc. |

Additionally, if a quote character is needed inside the NextBASIC command sequence, it can be escaped using the backwards slash character as follows: \"

Wildcards can be used to replace parts of a file type: * for the remainder, ? for only one character.

Browser.cfg can be edited using any standard text editor. More information about the Browser and how to configure it can be found by launching its guide file with:

```
guide browser
```

The Command Line

The NextBASIC editor is excellent for editing large programs, however for single use commands like the ones for file management or the dot commands we have been examining on a case-by-case basis, it can be a bit cumbersome to use, especially since the underlying NextBASIC string will appear after every other command. A bit less so, NextZXOS includes a special version of the NextBASIC editor, that hides (but does not erase) any NextBASIC program that you may be editing and offers an uncluttered view of the screen making it easier to enter commands directly to the operating system. Unlike other operating systems, the NextZXOS command line still gives full access to NextBASIC and doesn't include a prompt like the one available on CP/M which we'll examine a bit further. To access the Command Line interface, press EDIT to bring up the NextZXOS menu, select Command Line and press ENTER. While in the Command Line interface you have the op-

¹⁸ This functionality is much improved with dot commands that can not deal with LFNs.

how to change how many columns are displayed by either again calling up the *NextZXOS* menu with **EDIT** and selecting the 32x64/65 entry or by directly giving the **SPECTRUM CHR\$** command that can change the columns displayed immediately. See the **SPECTRUM CHR\$** entry previously in this chapter for details of usage.



ROM Cartridge Loaders

For users of ZX Interface 2, Ram Turbo, Dendyator and compatibles, *NextZXOS* introduces the ability to load ROM cartridge based software directly from the *More...* submenu and selecting the interface option. Since the ZX Spectrum Next starts with the expansion bus disabled, it provides a quick way to type the appropriate commands to load either 48K or 128K ROM based software as well as apply all the necessary settings to ensure maximum compatibility of cartridge based software. All you have to do is select the appropriate option, *NextZXOS* will make the necessary adjustments, enable the bus and load the software.

48K BASIC

The **48K BASIC** menu, located in the *More...* submenu, turns your ZX Spectrum Next into a standard 982 ZX Spectrum, with a twist. First of all, according to the *Next* personality you have selected during boot, you may have full key only, 'Looking Glass' (instead of token) or the keywords you see printed in your ZX Spectrum Next's keyboard (single-key only, ZX Standard). Additionally, you have access to all the ZX Spectrum Next's additional features although not from BASIC. Finally, you have access to your SD card via the dot commands we've already discussed. You can also reach 48K BASIC using the **SPECTRUM** command as discussed in a previous section.

128K BASIC

The **128K BASIC** menu, located in the *More...* submenu, turns your ZX Spectrum Next into a 985 ZX Spectrum 128K with the extra hardware of the Next available to it. Unlike the 48K option discussed above, the dot commands do not work as esxDOS requires a 48K BASIC (the so called 'USR0 mode').

ZX80 and ZX81 BASIC

This is a convenient way to access the ZX80 and ZX81 emulators by Paul Farrow without having to launch a separate personality on boot. There's no way other than 'reset' to come back from the ZX80/81 emulators.

NMI Menu

While in *Next* mode, pressing the **NMI** button will launch the *NMI* menu, which provides a list of user functionality to your ZX Spectrum Next. The *NMI* menu takes 'SingleStep' back to an expansion interface called *Multiface*. *Multiface* allowed users to pause a program and 'break into' it, create snapshots of the system's memory which upon reload, placed the machine in the same place they were, and 'running' the specific program they were at the point in time they were when they saved each snapshot.

The *NextZXOS* *NMI* menu offers, however, many more features over those of the original *Multiface*. We'll examine the most important ones of these below.

Upon loading, we can see the following entries in the menu.



Fig. 23 NMI main menu

Return - Hits off the NMI menu and returns you to whatever you were doing prior to pressing the NM button

Snapshot - Produces a snapshot of any legacy software that's currently running. It automatically recognises if it's a 48K type or 28K type of software and adjusts the snapshot type produced accordingly

Screenshot - Produces a screenshot of whatever is in any of the layers' screen memory areas and prints (to a ZX Printer or compatible) a JLA 'Layer 0' screenshot - also saves and restores the current palettes

TAP Files - Manages the redirection of input and output to drive (t)ape to virtual-tape files (tap) as well as browses their contents (in essence a shortcut to 'tapein', 'tapeout' and 'istap' we've covered previously)

POKES - Manages and applies 'pok' files to running software. These are files containing known workarounds and patches to specific applications - used mostly for games (for in-finite lives etc)

Debug tools - Gives access to maybe the most powerful set of features in the entire suite. A 'next' Register and Z80 Register status browser, a memory map and bank browser, the ability to set breakpoints in memory, to intercept running code as well as a banked memory save tool

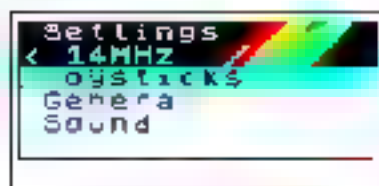


Fig. 24 NMI Settings



Fig. 25 NMI Joystick Settings



Settings - Allows easy modification of hardware settings on-the-fly. Fig. 27 NMI General Settings from the ones available on the configuration menu to the ones that are more nuanced (like the type of DMA chip in use or the machine timings used in the specific personality) which aren't always available through the standard configuration (Fig. 24 through 27)

Keymap - This is a duplication of the `keyhelp dot` command and provides a quick on-screen legend of the keyboard 'tokens' (for the 48K mode) which is particularly useful if using a board-only Next or a PS/2 keyboard

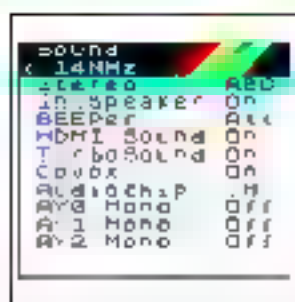


Fig. 26 NMI Sound Settings

ports and suggested features.

as needed as can be seen in the next figure.

The NextZXOS folder structure

The following table lists the files and folders that need to be present on an SD Card.

NextZXOS folder structure

are the particular settings you require for your machine.

ware the particular settings you require for your machine.

to install and unzip.

ever will be limited.

NextZXOS dot commands

erate esxdos errors in the basic system, either canned ones or custom ones.

erate esxdos errors in the basic system, either canned ones or custom ones.

System Error

your own.

System Error

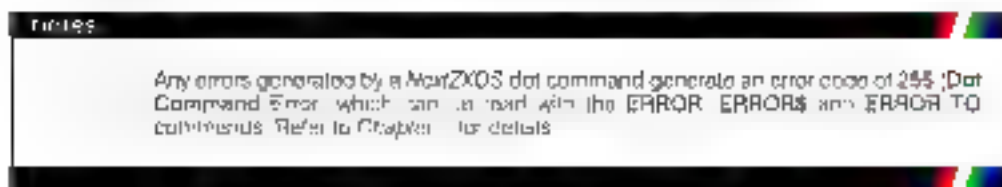
\$

NextBASIC so \$ allows execution of a dot command accepting any parameter passed as a string, thus enabling full integration of dot commands in NextBASIC.

bas2txt.txt txt2bas

each keyword occupies one token (see Appendix A for these values).

| | |
|---------------------------------------|--|
| | That further means that it's only machine and not human-readable other than from within the <i>NextBASIC Editor</i> . These two dot commands allow <i>NextBASIC</i> to be exported to a text file to be edited by a more specialised programmer's editor, or shared with other non-Sinclair computers and imported back in a form that the <i>NextBasic Editor</i> can understand. |
| browse | One of the nicest features of the <i>Browser</i> is its built-in file dialogs. browse allows these to be used within your <i>NextBASIC</i> programs and pass the selected file to a string variable in your program saving immense amounts of time from programming menu-based navigation. |
| defrag | <i>NextZXOS</i> provides a streaming API which can be used for audio or video. If the files however are not defragmented, streaming is interrupted. defrag solves this problem rearranging the file in question to be in one, continuous piece. |
| .editprefs .browseprefs | <i>NextZXOS</i> 's native customisers for the editor and the browser. |
| guide | gde is the official documentation <i>NextZXOS</i> file format and <i>NextGuide</i> is its viewer. It's a hypertext viewer partially compatible with the Amiga Guide Format. |
| install, uninstal | These are the dot commands to install and remove drivers like for example the <i>mouse driver</i> from the system. <i>NextZXOS</i> provides a driver API which you can use to write your own drivers which is used in conjunction with the new DRIVE-R command. |
| lfn | This is a very special use case command. Its sole purpose is to return the long file name for a short (8+3) filename. lfn does not work on <i>IDEIOS</i> , <i>+3DOS</i> drives, or rather it does work but returns the same name as <i>+3DOS</i> drives only accept 8+3 filenames. |
| mem | Returns the free memory for <i>NextZXOS</i> and <i>NextBASIC</i> use. |
| nextver | Assigns the current version of <i>NextZXOS</i> to a variable we specify. |
| unzip | Native decompressor for zip archives. |



Modifying the startup Autoexec.bas

NextZXOS provides you with a very fast way to set up your *NextBASIC* and *NextZXOS* environment upon boot by using commands stored in a special file called **autoexec.bas** located inside the *c:\nextzxos* folder. The same rules apply as with regular **SAVE**, meaning you will need to give a **LINE** parameter to save it before it can auto execute. If you omit the **LINE** parameter, the commands will auto load upon boot but won't execute. For example to set up a red background with bright white letters upon boot:


```

10 SPECTRUM PAPER 2 SPECTRUM
   BRIGHT 1 SPECTRUM INK 7
20 ERASE REM ERASES ALL LINES

```

Then

```
SAVE "c:/nextzxos/autotexec.bas" LINE 10
```

Reset and "magic"

CP/M

The ZX Spectrum Next supports running *CP/M Plus* (also known as *CP/M 3.0*) an operating system available for many microcomputers in the late 1970s and early 1980s.

CP/M provides a command-line environment similar to MS-DOS. A huge amount of software was available for it including programming languages, both interpreted and compiled, word processors, such as the well-known *WordStar*, spreadsheets, databases, utilities, text-based games and much more.

The ZX Spectrum Next runs *CP/M Plus* using a specially-written BIOS (Basic Input/Output System) which gives it a 80 x 24 text-based terminal supporting full colour.

To run *CP/M* you need to call up the *NextZXOS Startup menu*, go to the *More* submenu and select the *CP/M* option or from *NextBASIC* or the *Command Line* use the dot command *cpm*.

Any software compatible with *CP/M-80*, *CP/M 2.2*, *CP/M 3.0* or *CP/M Plus* will work on the

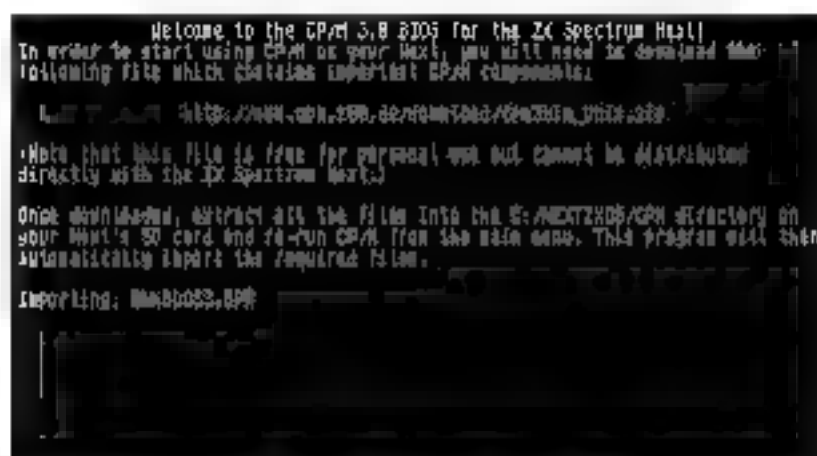


Fig. 28 Initial *CP/M* setup procedure

ZX Spectrum Next's flavour of *CP/M* except *CP/M-86* software (which requires an Intel x86 processor) and *CP/M-68* software (which requires a Motorola MC68k class processor).

Please note that *CP/M* graphical applications requiring *GSX* cannot be used at the moment, although support for those is under consideration. This is not affecting software availability considerably, as there is very little software requiring *GSX* most *CP/M* software was text-based.

Getting started

Before you can use *CP/M*, *NextZXOS* will need to prepare it. This is a process that happens automatically just once. You will need to access your *NextZXOS Startup Menu*, then from the *More* option, select the *CP/M* submenu. *NextZXOS* will start working on its own and once it finishes it will exit back to *NextZXOS*. From then on, every time you choose the

CP/M option from the *More* submenu in the *Next2XOS Startup Menu* or type *cpm* in the *NextBASIC Editor* or the *Command Line* will take you straight into CP/M (Fig. 29).



Fig. 2a. © Spectrum Next property, Inc. All rights reserved.

Commands

CPIM is operated by typing commands at the prompt **A>**. One of the most useful commands is **DIR** which works much in the same way that **CAT** works in *NextZXOS*.

Typing

DIREA

will show a list of all the files on the current drive or the drive specified. Initially you will just have drive A: available, but more can be set up. Drives A: to P: can be used using the `format` command in NexiaXOS as per the instructions provided earlier. So that you can keep different programs on different drives.

Any filename shown by DIR which ends in .COM is itself a command and can be executed at the prompt. You will have noticed there are a lot of .COM files to try. Another useful one is

HELP.COM

which provides help and information on all the standard commands and utilities provided with CP/M. Note that you do not need to type the .COM part at the time. CP/M will find the appropriate command and execute it without having to type its extension. In other words its file type. So to call up **HELP.COM** you could just type

HELP

Commands are also case-insensitive, so it doesn't matter if you type them in lower or upper case or a mix of both; all versions of `HELP`, `help`, `hELP` and `HeLP` will call the exact same program!

In the CP/M distribution that comes with *NextZXOS*, there are a number of commands specific to the ZX Spectrum Next. These include:

| Command | Description |
|----------|--|
| UPGRADE | Upgrades your installation of uP/M from the latest version available on your Suixcd. |
| TERMINFO | An interactive demonstration of the terminal facilities provided on the ZX spectrum. Next. |
| EXIT | Exits from uP/M and returns to NeoZX.OS. |

| Command | Description |
|----------|--|
| COLOURS | Changes the colour scheme |
| TERMSIZE | Changes the default terminal size (up to 80 x 32) |
| IMPORT | Imports files from your NextZXOS c: drive (or other FAT drives seen in the NextZXOS browser) |
| EXPORT | Exports files to your NextZXOS c: drive (or other) |
| ECHO | Sends text messages sequences to the terminal |
| NEXTREG | views (or changes) ZX Spectrum Next hardware registers (use B: your own disk) |

Typing the name of these commands will give some more information on how to use them.

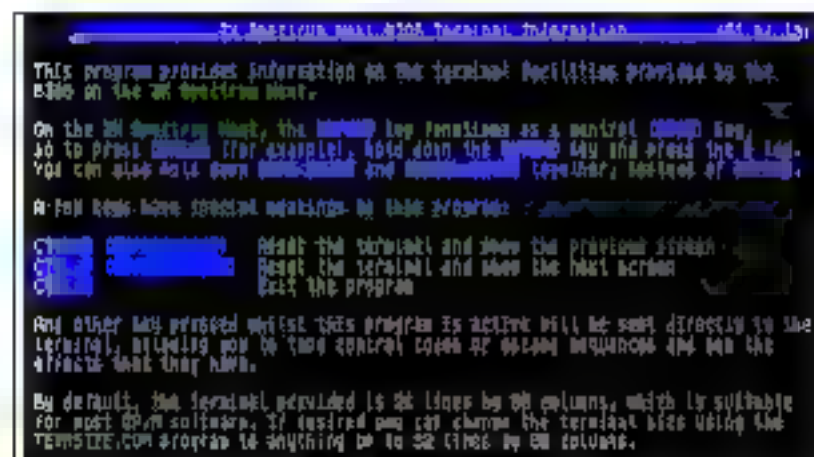


Fig. 10 TCBMINFO output

Drives and CP/M

CP/M on the ZX Spectrum Next cannot access the standard SD card drive c: (or other drives you may have due to having additional SD cards inserted) for example. This is because CP/M directly accesses disks at a low level and is incompatible with *FA* filesystems.

Therefore, on the ZX Spectrum Next, CP/M uses *virtual disk files*. These can either be *.p3d* files (created by the *mkdata* *dot* command) or *.dsk* files (images of standard ZX Spectrum +3 disks).

You can access multiple disk images at once in CP/M. To do this, simply create additional files with *mkdata* using the same naming scheme, eg. at the NextZXOS command line type the following:

```
mkdata /nextzxos/cpm-b.p3d
mkdata "/nextzxos/cpm-c.p3d"
```

When you next use CP/M you will have drives A:, B: and E: available. Note that you can have a drive C: in CP/M if you wish, but this is not the same as the c: drive used in NextZXOS.

Up to 15 *virtual disk images* can be used at once by CP/M and they can be mapped to any drive A: to P: simply by naming the files in any of these ways:

```
c:/nextzxos/cpm-X.p3d
c:/nextzxos/driv-X.p3d
c:/nextzxos/cpm-X.dsk
c:/nextzxos/driv-X.dsk
```


where X is the drive letter (from A to P) if you have created multiple files referring to the same drive letter. CPM will use the ones named **cpm-X** in preference to the ones named **drv-X**. It has no preference over **p3d** or **dsk** so if there's a **cpm-b.p3d** and a **cpm-b.dsk** then the first one in the directory will be used.

Note that NextZXOS will also automatically mount those drive images (except any image where X is c) when it starts up. You can view them in the Browser, press D to change drives, and copy files between them etc. NextZXOS will mount drv X files in preference to ppm-X files. You can also manually mount other disk images which don't follow the automatically mounted naming scheme. To do this, just press ENTER on the .p3d or .dsk file in the Browser.

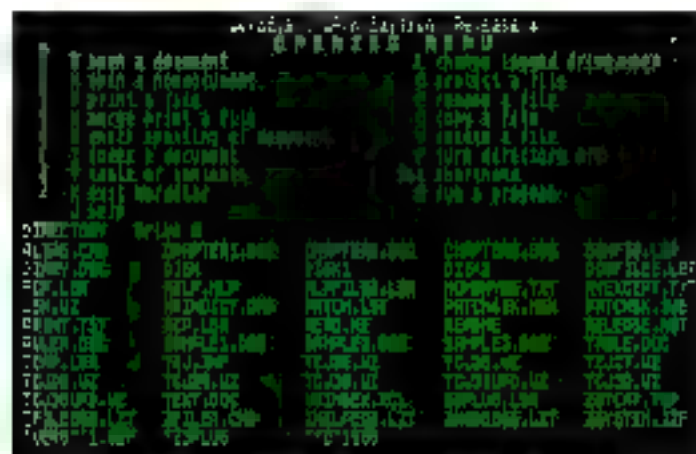


Fig. 31 ZX Spectrum Next CP/M running WordStar 4

Further information

There is a lot to learn about CP/M and a lot you can do with it. Some useful places for further information are listed below.

<http://www.com-z80.de> Contains a lot of manuals, documentation and software

In particular, the *CP/M 3 User Guide*, *Command Summary* and *Programmers Manuals* can be found in the following locations:

<http://www.cpm-z80.de/manuals/cpm3-usr.pdf>
<http://www.cpm-z80.de/manuals/cpm3-cmd.pdf>
<http://www.cpm-z80.de/manuals/cpm3-pqr.pdf>

User Guide
Command Summary
Programmer's Manual

A good starting point is also

<http://classiccomp.org/opmarchives>, which links to many more useful sites, collections of software, manuals, magazines and much more.

Preparing your ZX Spectrum Next for esxDOS

Other than NextZKOS, CP/M and +3e//EDOS, your ZX Spectrum Next supports natively one more Operating System called **esxDOS**. This is especially helpful when running East-ern European software as the preferred method of storage is using **FATDOS** which esxDOS supports natively. Unfortunately the copyright status of some parts of esxDOS prohibits its inclusion in the **System/Next**™ distribution but that doesn't mean you can't install it yourself: esxDOS can be invaluable for personalities other than the Next Name one as it provides older mode personalities with an easy way of managing FAT formatted SD cards. As is the case with NextZKOS, it also uses **FAT** as the primary filesystem and thanks to NextZKOS design, it can therefore co-exist on the same drive without clashes.

In order to install exxDOS you need to do a few things first:

- Go to www.esxdos.org and download either the latest version or the one whose rom comes with the System/Next distribution. For correct operation the minimum supported version is 0.8.6 beta 4.
- Using a PC, Mac or Linux machine, unzip the contents of the esxDOS distribution into a drive (connect the System/Next™ SD card onto the same computer and then do the following:
 - Copy the **Bin**, **SYS** and **TMP** folders into the System/Next™ distribution's root folder.
 - Copy the **ESXMMC Bin** file from the esxDOS root to **c:/machines/next/**.
 - Finally edit the **config.ini** file in **c:/machines/next/** to include esxDOS with the personality you choose. (Note that this doesn't apply to Next Native mode)

Here is an example that will modify config.ini to use esxDOS with the 128k personality (note that esxDOS will boot any 128K personality in what is called 'ISHC' mode, a special mode where the editor is 48K but all the 128k features are available. After you download the esxDOS distribution archive from the esxDOS site, unzip it and follow the instructions above. Then go to **c:/machines/next/** and using any text editor (for example *Notepad* under Windows) open **config.ini**. Locate the line reading:

```
menu=ZX Spectrum 128k,1.8,128 rom
```

and modify it as follows

```
menu=ZX Spectrum 128k 1.8 128 rom, esxmmc bin,<none>
```

Also, if you have an RTC chip installed, go to **c:/nextzxos/** and copy **RTC SYS** to **c:/sys/**. Save it, eject the SD card and transfer it to your ZX Spectrum Next. Upon boot, press **SPACE** and then using the cursor keys, locate the **ZX Spectrum 128k** line. Press **ENTER** and in a few seconds you'll see something like this:



Fig. 32 ZX Spectrum Next running esxDOS 0.8.6

That was it, you now have a functioning esxDOS installation for your 128K personality on your ZX Spectrum Next computer and the green Drive button on the left side of your computer will start functioning calling the esxDOS browser.

Chapter 20 Channels, Streams, Drivers and Windows

As we have seen thus far, NextBASIC can read data from the keyboard and controllers using `INPLT` and `INKEY$` and it can write data onto the display or a printer by using `PRINT` and `LPRINT`. However, these commands are really a form of shorthand designed to protect the user from some of the computer's more complex features.

To the `PRINT` command, for example, there is no difference between the screen and the printer. `PRINT "Mikayla"` really means *take the characters which make up the word Mikayla and send them somewhere else*. It's us (conveniently) to use the screen most of the time. Likewise, `LPRINT` usually sends data to the printer. In fact, what these commands really do is to send data to one of a number of *channels*.

Channels

A channel is the pathway to the computer's input and output devices and on the ZX Spectrum Next, they are designated by a letter. These are:

| Designator | Direction | Description | Default Stream | Default Status |
|------------|---------------------------|-------------------|----------------|---------------------|
| k | Input/Output ¹ | Keyboard | 0.1 | Open |
| s | Output | Screen | 2 | Open |
| p | Output | Printer | 3 | Open |
| i | Input | File (input) | | Closed ² |
| o | Output | File (output) | | Closed |
| u | Input/Output | File Update | | Closed |
| v | Input/Output | Variable | | Closed |
| m | Input/Output | Memory | | Closed |
| d | Depends | Driver | | Closed |
| w | Input/Output | Windows | | Closed |
| r | Internal | Internal Use only | N/A | Open |

Table 3: NextBASIC Channels

To access a channel, it must be open. Opening a channel makes it ready to receive or produce data. A channel is opened by connecting it to a stream. From NextBASIC, you would use a command like

```
OPEN #4, "K"
```

which means *connect stream 4 to the keyboard channel*. As evidenced by the table above, we go by the direction of data flow: there are three types of channels: *Input*, *Output* and *Input/Output* or *Update*.

However, we can better classify channels by device type. We have *Screen*, *Keyboard*, *Printer*, *File*, *Memory*, *Variable*, *Windows* and *Driver* channels. Let's examine them according to the device type however, as this affects what types of commands we can use with them and how.

The *Screen Channel* deals with everything that goes on the screen. It is the simplest of all channels and most of its characteristics have been covered in Chapter 15 already. It is already opened and connected to stream #2. In fact you can substitute any `PRINT` command with `PRINT #2` and it will work in the exact same way as a regular `PRINT` command.

¹Outputting data to the keyboard might seem a bit weird, but it's important to consider that the controller does the heavy screen I/O. INPLT does not display the characters, it becomes clear why.

Windows are defined by their top line (0-23), leftmost column (0-31), height (1-24), width (1-32) and optionally character size (3-6) and character set address. If no character size is specified, the default is 8. If a character set address is given, then this is used instead of the built-in fonts. This allows you to use fonts such as those provided with art programs and adventure games. Due to their complexity, we'll devote an entire section to Windows after we discuss streams and the commands with which we use them.

Streams

Streams³ are convenient ways for the computer to switch between channels by referring to them as numbers. This idea makes it possible to write programs that can send information to any device without having to use different commands. There are 16 total available streams, numbered 0 to 15. 4 streams, 0 through 3, as seen on the table above, are already opened to channels **k**s and **p**. Note here that many streams can be attached to a channel depending on what we want to do.

Using Streams

All the above might seem complicated, and you may well wish to stick to the standard **PRINT** and **INPUT** commands. That's why they're here after all. Even these commands, however, are just shortcuts to their complete versions that also include a stream number and the benefits of using channels far outweigh their perceived complexity.

Stream control commands

Since it's now evident that any device on the computer that accepts input or produces output is really a channel, it's easy to realise that we have been using streams all along: we've already visited **PRINT** and **LPRINT** (which are really the same command), used **INPUT** and **INKEY\$** and (as yet) we've used **LIST** and **LLIST** (which also are the same command). All the above have versions which include a # (hash) followed by a current stream number, so we are already halfway there.

Apart from these and **OPEN #** we saw in the channels section above, the following commands are available for working with streams: **CLOSE #**, **DIM #**, **DIM # TO NEXT #**, **NEXT # TO POINT #**, **RETURN #**, **TO GOTO #**, **TO** and **COPY TO #**, **CAT #** and **PWD #**. We'll examine them all below.

OPEN #n, channelspec

where **n** is the stream number and **channelspec** is a string that can be any of the following (capitals or lower case letters may be used): opens a stream and attaches it to the channel defined by **channelspec**.

| String | Description |
|-------------|---|
| "k" | The standard input channel (keyboard and lower screen). Streams 0 & 1 are normally set to this channel. |
| "s" | The standard output channel (main screen). Stream 2 is normally set to this channel. |
| "p" | The standard printer channel (serial or parallel). Stream 3 is normally set to this channel. |
| "<>filepec" | This opens an input-only stream to an existing file. If the filename is at least ten characters long and an emulating system will be assumed. For file names require the "<>" as otherwise file will be assumed to be standard channel names. |
| ">>filepec" | This creates a new file and opens an output-only stream to it. |

3 On other versions of BASIC, streams are called channels and channels are called devices. This may be a bit confusing, as we already have a different version of BASIC. For versions however are identical, the same.

4 Altering streams will change the behaviour of the system and should be used with care.

| String | Description |
|---------------------------------|--|
| "<n>filepec" | This opens an existing file and opens an input/output stream to it |
| "<n>address length" | This opens an input/output channel to the memory area at address length |
| "<n>var" | This opens an input/output channel to the variable var which must be a character array with a single dimension large enough to hold everything that will be output to it from BASIC |
| "<n>file control and size, var" | This opens an input/output channel to a text window on the screen, starting at character position file ctrl with a height of character rows and a width of character columns. Typically a character width is size 3-5, it may be specified. It does not affect the character details of the window which are always specified in full-width characters. A user-specified character set may also be specified, located at address ctrl. See the Windows special section for details |
| "<n>driver name>[driver spec]" | This opens a channel to driver name whose data flow direction is controlled by the driver addresses. Driver spec is optional and depends on the driver (if needed or not) |

Table 20-1 BASIC # channels per setup strings

Here are some examples:

- OPEN #4,"c:>test.txt"** Creates a file named test.txt on virtual disk drive B, and opens an output-only channel to it, connected to stream 4
- OPEN #5,"stuff"** Opens an existing file named stuff on the default drive and opens an input-only channel to it, connected to stream 5

Once a stream is opened, it can be used with the standard **INPUT #** and **PRINT #** commands, as well as the additional pointer commands. Before we get into those, we should just first mention

CLOSE #n

which closes the previously opened stream #n. If n is a stream between 0 and 3, then the default channel for that stream (k, s or p) is reattached to it. Note that attempting to **CLOSE** a stream that hasn't been opened will not produce an error; instead it will exit gracefully with OK, 0. For example:

- CLOSE #4** Closes the channel attached to stream 4

Streams, and especially those opened to large files, can be very long to navigate in a serial manner. Imagine having a file that's 10 Kbytes long; you would have to iterate through 102400 characters to read the very last one byte. For that reason, NextBASIC maintains pointers to the position you're located within a stream, how long the stream is (in characters, bytes), the ability to move these pointers to any location within a stream, and finally the ability to read one byte from the current pointer position from that stream. The commands and functions to do that are called *Pointer Commands* and are the following **POINT #** and **RETURN # TO DIM#** and **DIM # TO GO TO #** and **NEXT # TO**. Let's visit their syntax below:

POINT #n

RETURN #n TO %|var

This command returns the current position of stream n to the same as the **RETURN # TO** with the exception that no variable assignment is done to the resulting value if the **RETURN** variant is used; then it also stores it in variable var. The variable can be an integer one which means that it will accept safely positions of up to 65536 bytes within the stream, or a maximum value of 65536 as position 0 is the very first position within a stream. Do not use integer values to plan on accessing streams larger than that; if you don't use the **TO** variant, however, you can use it as part of the regular expression evaluator.

DIM #n

DIM #n TO [%]var

This command returns the size (in characters or bytes) of stream *n*. Whatever applies to the **RETURN # TO** variant of **POINT #** applies to the **DIM # TO** as well. Variable *var* stores the size of the stream. As with **RETURN # TO** above, *var* can be an integer variable in which case the same warning as with the previous section applies.

GO TO #n, [%]pos

This command sets the current position of stream *n* to position *pos*. Let's see how the previous three commands all tie together by experimenting with **browser.cfg**.

```

10 OPEN #4 'nextzxos/browser
   .cfg
20 REM '1)' is optional since
   the filename is longer
   than 1 character
30 DIM #4 TO %a REM Get
   filesize and put it in %a
40 RETURN #4 TO %b REM Get
   current location and put
   it in %b
50 PRINT 'You're in byte ',
   %b, ' of ', %a
60 GO TO #4, %a/2 REM Move
   to the middle of the file
70 RETURN #4 TO %b REM Get
   current location and put
   it in %b
80 PRINT 'Now, you're in
   byte ', %b, ' of ', %a
90 CLOSE #4

```

NEXT #n TO [%] var

This command gets the next character of input from stream *n*. As with **POINT #** and **DIM #** if used with the **TO** modifier it also stores it in the variable *var*. If used on the standard keyboard channel, this is similar to the **INKEY\$** function, except that it always waits for the next character to become available (ie on the keyboard channel it waits for a keypress). Using an integer variable here, is safe as the command gets one character at a time ergo the byte so its value will never exceed 255.

You can use this command instead of **INPUT #** on all channels that accept input otherwise they're very much identical in function.

Try this little program which will turn your ZX Spectrum Next into a typewriter:

```

10 NEXT #0 TO X
20 PRINT CHR$(X),
30 GO TO 10

```

Alternatively you could change line 10 to


```
10 x=NEXT #0
```

which does the same thing

COPY *filespec* **TO** *#n*

We've seen this command sequence before in a *shortcut* which did not include a stream number but rather a keyword: **SCREEN\$**. In that case *n* is the stream-to-channel *s* which by default is #2. When used with a stream number **COPY TO #n** can be used to transfer the contents of a file to a stream. For example to write the extended version of **COPY "c:/readme.md" TO SCREEN\$** we should type

```
COPY "c:/readme.md" TO #2
```

When NextBASIC is running it has four streams normally open. Streams #0 and #1 are connected to the keyboard channel *k* and are used by **INPUT** and **KEY\$**. Stream #2 is connected to the screen (channel *s*) and is used by **PRINT**, **LIST**, **CAT** and **PWD** commands in other words that print something to the screen. Stream #3 is connected to the printer channel *p* and is used by **LPRINT** and **COPY** without parameters. All of these commands can be redirected to use another device by including a # followed by an open stream number *s*.

```
PRINT #1 "This is the lower screen"
```

will print the message on the lower screen while

```
PRINT #3 Who needs LPRINT, Royals?
```

will use the printer. Conversely **LPRINT** can behave like **PRINT** and typing

```
LPRINT #2, "Are you confused yet Roy?"
```

makes **LPRINT** #2 do what **PRINT** normally does

Notes

INPUT # may be used with other channels other than *k* and *w* such as *file* or *memory*, *m*, and *device*, *d*, channels. In these cases it is advisable to avoid any accidental outputs to the channels, by not using any prompt strings and by using only the semicolon as a separator. In most cases you will want to input a string using the **IN\$**. (See Chapter 4) *mod* or *as* without using the data in the file (or other channel) would need to be surrounded by quotes.

The Variable and Memory Channels

In the previous chapter we've examined a special dot command **\$.** that allowed NextBASIC to talk to any dot command not made specifically to interact with it. The variable and Memory Channels can be seen as facilitating the reverse flow of information to get information from the outside world into NextBASIC. They both involve reserving some space beforehand to accept the input but they differ in the sense that the former can be moved anywhere in memory (as variables could be stored anywhere) while the latter is a fixed location (which makes it more suitable for use by machine code programs). You may remember the series of commands we used to get the output of **PWD** in Chapter 19 or time in Chapter 17. Let's remember them quickly

```
DIM d$(255) OPEN #2, v>d$+".cd" verbose
CLOSE #2 PRINT d$
```

and


```
DIM t$(100) OPEN #2, 'v>t$':,TIME CLOSE
#2 PRINT t$
```

but now that you know a bit more about streams, should that even work? The answer is yes, as it's designed to work that way. Most dot commands that produce textual output in a 'legal' way (that is without circumventing NextZXOS) will attempt to output content on stream #2. By opening stream #2 to the variable channel and then executing the command whose output we wish to capture, we're performing a temporary redirection of the screen stream to the variable channel. Then, once we close the stream again, as the system is designed to do, it resets it to its default channel 5 and reopens it. Obviously if a program does not use the inbuilt NextZXOS and NextBAS/C routines to produce output, this will produce nothing. The example below shows a more 'traditional' way of using the variable channel by using the inbuilt facility of a command 'CAT ASN' in this case, to output to a different channel.

```
10 DIM a$ 1000)
20 OPEN #8 'v>a$
30 CAT #8 ASN
40 RETURN #8 TO 1
50 PRINT 'Assignment length
   is ',L, ' chars'
60 PRINT 'List is '
70 PRINT a$( TO L)
80 CLOSE #8
```

notes

If a stream operation fails (like in the example above), the stream will not automatically close. It is therefore a good practice to close all your programs that operate in a stream with a `CLDSP #` (which is actually performing an `OPEN #` operation for the first time). It's also ever useful programming practice to include `ON% ERROR` error-trapping. In every stream operation, especially the ones that operate in File Channels, as a lot of things can go wrong while working with files and channels in general (e.g. running out of data, or your reserved memory area was smaller than the one you should have reserved etc).

As you can see line 40 also demonstrates the use of a pointer command in the variable channel – you do not reserve enough room for the sake of displaying the results (change the size of `a$` to just 10 characters from the 1000 it has – you will receive an `%End of File` error at line 30).

The memory channel operates in a very similar manner, once you reserve the space, you open it and dump the output to it. Let's modify the above program to use the memory channel.

```
10 CLEAR 209999
20 OPEN #8, 'm>00000 1000"
30 CAT #8 ASN
40 RETURN #8 TO 1
50 PRINT 'Assignment length
   is ',L, ' chars'
60 REM perform some magic
   here via MC
70 FOR f = 0 TO L-1
```



```

80 PRINT CHR$(PEEK 30000+f)),
    REM print the 4 first
    bytes you stored in memory
90 NEXT f
80 CLOSE # 5

```

Installable device drivers and Driver Channels

As mentioned in the previous chapter, *NextZXOS* allows for installable device drivers. A maximum of 4⁶ of those can be installed.

These are mainly intended for use as software that allows access to external or internal peripherals such as printers, i.e., network devices etc. but can also be used for other purposes such as a potential *NUL* driver which does nothing. (The notion of a device that does nothing is a bit peculiar but it has its uses in computing.) As mentioned in Chapter 19, to install or uninstall a driver, you need to use the following *dos* commands respectively:

```

install drivename
uninstall drivename

```

where *drivename* is the name of the file which contains the code for each driver. For example the *WiFi* driver for the ESP chip that your *ZX Spectrum Next* may have come with or you may have installed yourself is *espa1.drv*.

The documentation that comes with the driver will describe how to use it. Some drivers for example may make use of the new **DRIVER** command. This has the following form:

DRIVER *driverid*, *callid* [*n1*,*n2*] [TO *var1*, *var2*, *var3*,...]

where *n1* and *n2* are optional values to pass to the driver and *var1*, *var2* and *var3* are optional variables to receive results from the driver call. The individual **DRIVER** commands that you can use, depend on each device driver and they will also be in the driver's accompanying documentation.

Driver Channel support

Some drivers can support input/output via streams and the Driver Channel *d*, so the documentation will describe the exact format it supports. Generally speaking however, in order to open a stream to channel *d*, you will be using one of the following command variants (assuming the driver id is ASCII X):

```
OPEN #8,"d>X"
```

which opens stream #8 to simple driver channel for device X

```
OPEN #8,"d>X>string"
```

which opens stream #8 to channel *d* as described by *string* on device X

```
OPEN #8,"d>X,p1"
```

which opens stream #8 to channel *d* as described by numeric value *p1* on device X

```
OPEN #8,"d>X,p1,p2"
```

which opens stream #8 to channel *d* as described by numeric values *p1* and *p2* on device X

6 This numbering changes in subsequent versions of *NextZXOS*.

To close the driver's stream you will use a standard **CLOSE #** command (in the examples above that would be **CLOSE #8**).

Once the driver's channel is open you can use any of NextBASIC's stream input/output or pointer manipulation commands (if these are supported by the loaded driver. Usually each driver's documentation should describe what can be used).

A good example of using the driver channels can be found in the documentation for the ESP/Wolf driver by Jim Gilberts included in the c:\docs\extra-hw folder of the System/Next™ distribution. You can see there for example that talking to the internet via NextBASIC can be as simple as

```
OPEN #4,"d>N,TCP,145 239 200 34 80"
```

which will open a TCP connection to port 80 on speenext dev.

Windows

NextBASIC offers the ability to create and manipulate text "windows" on screen via its Window Channels. This allows for immense flexibility in manipulating textual output, going beyond what simple PRINT commands can.

System Windows vs User Windows

When we talk about Windows, we're really talking about two kinds: System and User Windows. The former are created and managed by NextBASIC while the latter are created and controlled by the user. By default 4 System Windows are created, one for each Layer other than 0. These are full screen and are used to produce output through the standard channels and only a few parameters of these can change. Size always remains the maximum possible.

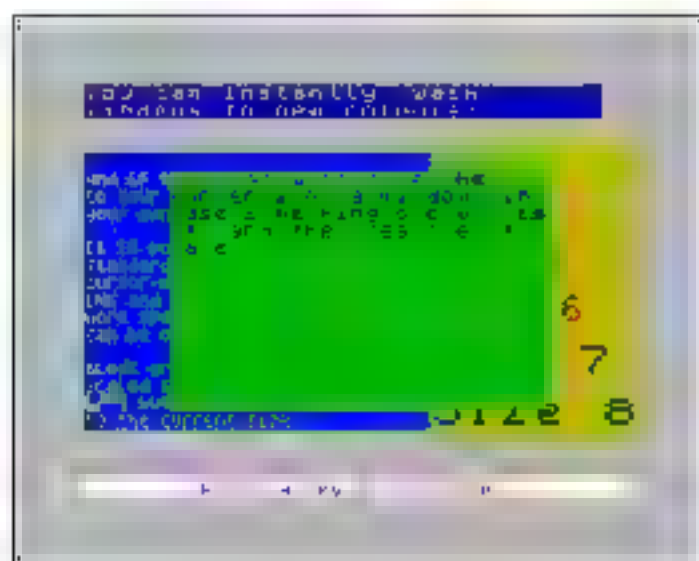


Fig. 33 NextBASIC Text Windows

User Windows on the other hand can have varying sizes and can be defined anywhere in the screen. From now on we'll refer to System Windows as SW and to User Windows as UW. If no designation exists, then the discussion applies to both types.

Defining User Windows

User windows are defined by their top line (0 to 23), leftmost column (0 to 31), height (1 to 24), width (1 to 32) and optionally by character size (3 to 8) and character set memory ad-

⁸ Memory address refers to an address indicator within the max. memory map.

dress. If no character size is specified, the default is assumed which is 8 px wide. If a character set address is given, then this is used instead of the built-in fonts⁸; this allows you to use nice fonts such as those provided with art programs and adventure games.

The character size has no bearing on the way the window is defined, but it does affect the number of actual columns you have available. For example, the following defines a window the size of the entire screen, but because a character size of 5 is specified, the number of characters that can be printed in the window at any time is 24 x 51.

```
OPEN #5,"w>0,0,24,32,5"
```

When outputting via PRINT to windows, you can use many of the same control functions as you can with the normal screen. For example, `approxim`, `start` a new line, `comma`, `start` a new column, `TAB`, `AT`, `POINT`, `INK`, `PAPER`, `FLASH`, `BRIGHT`, `INVERSE`, `OVER`.

When first defined, windows are in *non-justified* mode, but they can be set to be *left*, *full* or *centre* justified. Note that in *justified* mode, some features and control codes cannot be accessed, so you may need to switch back to *non-justified* mode to use them.

A complete list of control codes follows in the table below. These codes can be sent to a window using PRINT followed by the CHR\$ function as we've already seen in *Chapter 14*. Note that it's always preferred to use standard PRINT, AT, NK etc. commands instead of control codes when using windows as they're usually easier to use than their control codes counterparts. Below is a list of all control codes that can be used while outputting to a Window Channel's stream.

NOTE that wherever there are sequential numbers they must be given using semicolon separated CHR\$ statements. For example:

```
PRINT CHR$ 29, CHR$ 2
```

| | Code | Description | |
|---|------|--|---|
| | | JW | SW |
| | 0 | Turn justification off | Increases the current character set width (can range from 3 to 8 pixels). A 0 moves the cursor to the start of the next line. |
| | 1 | Turn justification on | Increases the current character set width (can range from 3 to 8 pixels), and moves the cursor to the start of the next line. |
| | 2 | Save current window contents | Causes the save & character set to be replaced with the character set defined by the CHARS system variable. |
| | 3 | Restore saved window contents | Causes the save & 7 character sets to be regenerated. |
| | 4 | Home cursor to top left | |
| | 5 | Home cursor to bottom left | |
| X | 6 | Tab to left or centre of window (PRINT,) | |
| | 7 | Scroll window | |
| X | 8 | Move cursor left | |
| X | 9 | Move cursor right | |
| | 10 | Move cursor down | |
| | 11 | Move cursor up | |
| X | 12 | Delete character to left of cursor | |

⁸ A font is a collection of a stylised, graphical representation of characters. For the ZX Spectrum Next this follows the 808 pixel height of the 1024s and is a roughly 768 bytes long (defining 256 characters in the 7-bit Single ASCII format from 32 to 255). See Appendix A for a list of characters.

| | Code | Description | |
|---|-----------------|---|---|
| | | JW | SW |
| | 13 | Start new line (PRINT) | |
| | 14 | Clear window to system attributes | |
| | 15 | Wash window with current attributes | |
| ● | 16: n | Set INK n (where n = 0 to 7) | |
| ● | 17: n | Set PAPER n (where n = 0 to 7) | |
| ● | 18: n | Set FLASH n (where n = 0 or 1) | |
| ● | 19: n | Set BRIGHT n (where n = 0 or 1) | |
| ● | 20: n | Set INVERSE n (where n = 0 or 1) | |
| ● | 21: n | Set OVER n (where n = 0 or 1) | |
| × | 22: y, x | Sets cursor to pixel line y, character at column x (AT y,x). Position is specified in terms of character positions, depending on the character size currently set in 24, whether n is a n-high bit is supported. Double-width and double-high don't affect the coordinates, however. | |
| × | 23: nLow, nHigh | TAB n character n, where n is 1 to 34. If n is greater than 34, then number single tab. 255 graph is always 0. Otherwise, n is absolute as nLow, nHigh*256. | |
| ● | 24: n | Sets ATTR n (where n = 0 to 255) | |
| × | 25: yLow, xHigh | Changes the origin position to pixel coordinates x = 0 to 511 and y = 0 to 191 respectively, unless n may be changed to 2 mode. xLow and yLow position may be greater than 256. yHigh is equal to yLow + 191 while xHigh is an offset from the origin. The two pixel coordinates are 0 to 255, a range 0 to 191, at 256 bit, yLow is 0, xHigh is always 0 while, if resolution is > 256 pixels it may be 0 or 1. The coordinate is calculated as (xLow) + (xHigh*256). | |
| ● | 26: n | Auto-pauses every n character lines. After each n character lines have been scrolled out of the window, output will automatically pause until the SPACE key is pressed (the bottom left character in new window will be displayed) and on SPACE is being waiting to write new window has the clearing or less pause. If previous line have been scrolled out, auto-pause wait for character as typically, will be wait set on the right of the window. If set to 0, the default auto-pause is disabled. | |
| ● | 27: n | Fills window with character n. Attributes and cursor position are affected. | |
| × | 28: n | Sets double width (where n = 0 is normal width, where n = 0) | |
| ● | 29: n | Sets height n (0 = normal, 1 = double, 2 = reduced, 3 = double reduced). See Chapter 4 for details. | |
| | 30: n | select justification mode, where n = 0 = left justified, 1 = fully justified and 2 = right justified | changes the current character set width, n (can be 3, 4, 5, 6, 7 or 8 pixels), and moves the cursor to the start of the line. |
| | 31: n | selects whether embedded codes are permitted (0 = no, 1 = yes) or code | causes the same character set to be replaced with a character set defined by the ANSI system variable. |

Table 2 Window control codes

In the table above, on the column marked as J or S: X means ignored if used in justified mode. A to an ● means code can be used in justified mode only if the embedded codes setting has been enabled. For control codes normally ignored in justified mode, note that these will still be taken into account if you set them before entering justified mode.

User character sets

The default character sets are replaced using control codes 2, 3 or 31 in a system window. Any subsequent text printed in any window (which doesn't have its own user-defined character set) will use the new character set(s).

The system-defined character sets are partially shared: sizes 3 and 4 use the same set of 12 by the terminals 3 pixels are used for size 9), and similarly sizes 5 and 6. This should be borne in mind when replacing system character sets using control code 31.

8 - has no effect on user's or LINES

9 - ignored, also in Fullscreen - ignored, also in Fullscreen and Fullscreen/ANSI embedded

10 - ignored, also in Fullscreen - ignored, also in Fullscreen

Window input

Text windows support the **INPUT** command. If you use **INPUT #**, then a cursor is added to the window at the current position. You can then input any text desired using the left and right arrows to move along the text input so far, or the up and down arrows to move to the start or end of the text.

The **DELETE** key deletes the character to the left of the cursor, and the **ENTER** key completes the input. Up to 191 characters can be accepted into each input variable.

Window definitions

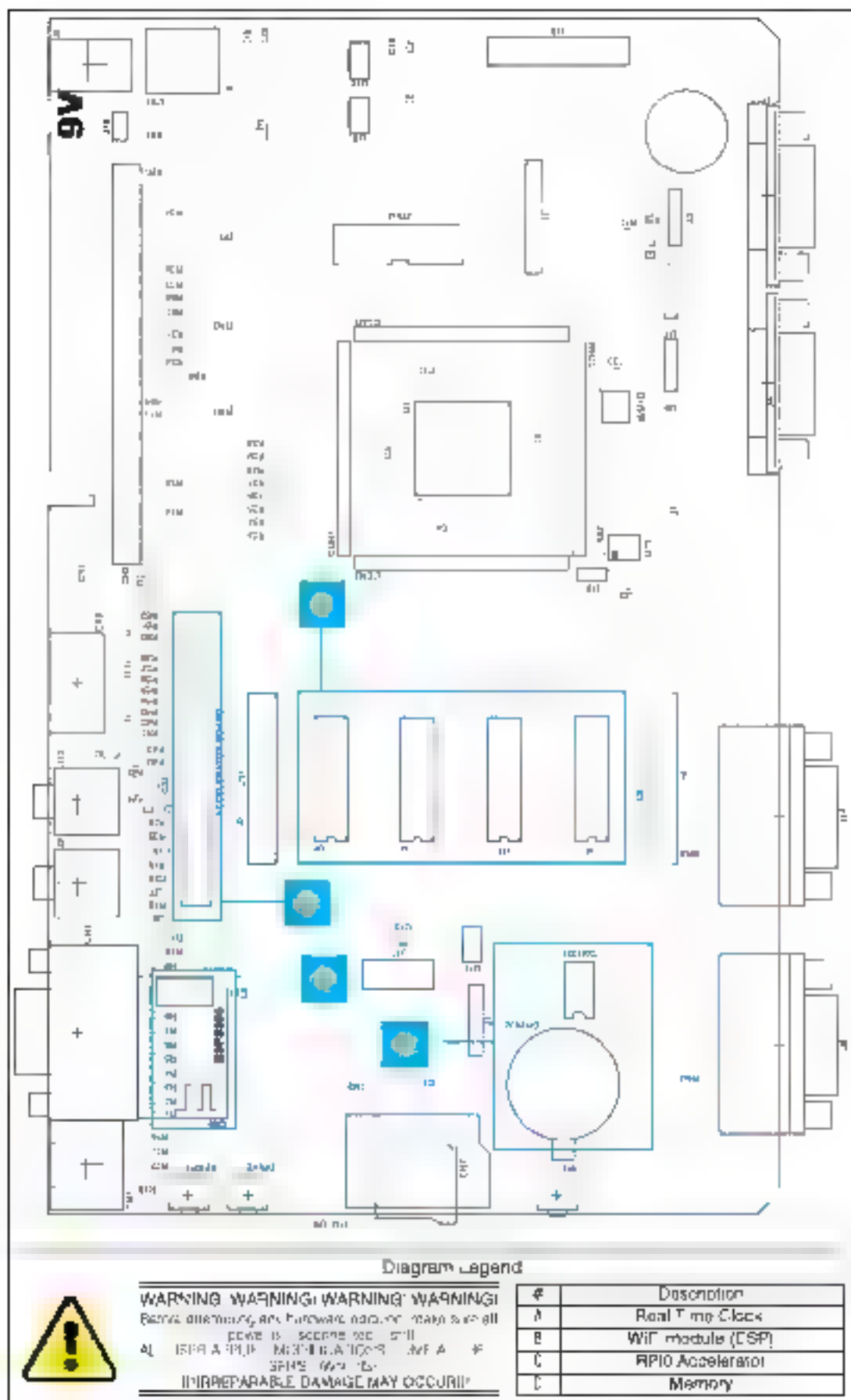
Since windows are defined using character squares so for example in LoRes, this means the maximum window size is 16 × 2, and not 32 × 24. In HiRes however, character squares are considered to be 16 pixels wide, so the maximum window size is still 32 × 24 pixels.

Memory constraints

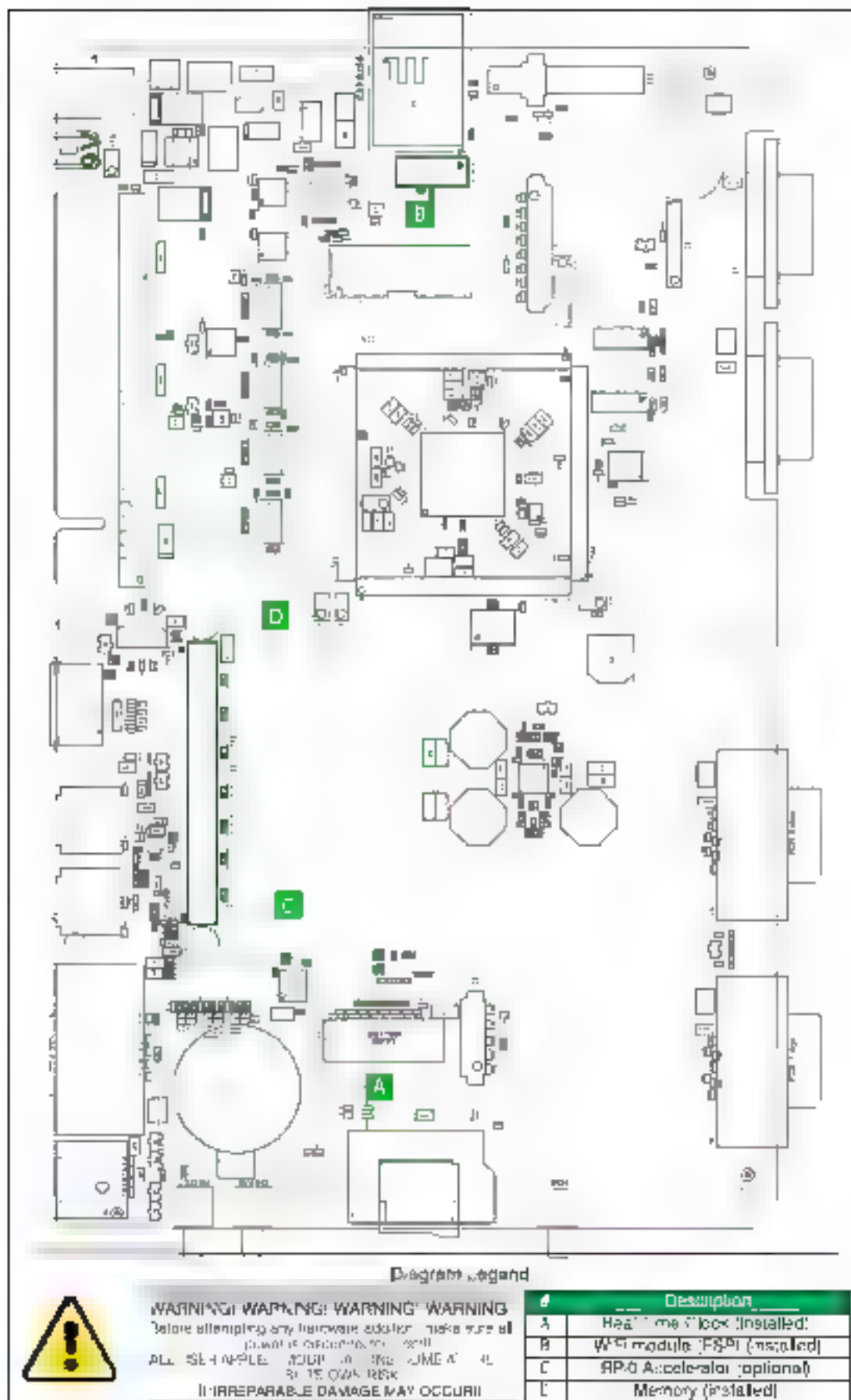
It should be noted that saving/loading window contents (only available on user windows) is a costly operation. The amount of memory required for each character square is:

- 9 bytes (Layer 0)
- 16 bytes (Layer 1 HiRes or HiColour)
- 64 bytes (Layer 1 LoRes or Layer 2)

For example, a 6 × 10 window in Layer 2 requires 6400 bytes of available memory for saving its contents.



The ZX Spectrum Next Issue 2 Mainboard with optional equipment locations



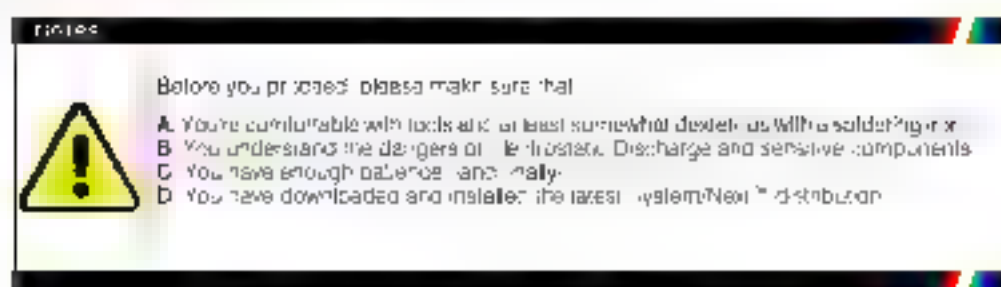
The ZX Spectrum Next Issue 4 Mainboard with equipment locations

Chapter 21 Optional Features

Overview

Depending on the model you have and the Kickstarter you participated in, your ZX Spectrum Next may have one or two types of mainboards: issue 2 or issue 4 (See appropriate figures in the beginning of this Chapter). If you have an issue 2, the rest of the chapter is for you, as many of the standard features on an issue 4 used to be optional on the issue 2. These are *RTC* hardware, a *WiFi* module, *ESP*, extra *RAM* and the *Raspberry Pi Zero* (*RPi0*) accelerator.

For the other hand, you have an issue 4, the only feature you may be interested in is the *Raspberry Pi Zero* accelerator. The following sections will describe how to install and use them. Remember that modifying your ZX Spectrum Next carries a number of risks and that if you are not careful, you can damage your machine.



Installation (for issue 2 mainboards)

Most additions are very easy to install with the exception of the *Real Time Clock* module and *RPi0* accelerator. Installation of the former requires soldering a number of parts onto the board and should be undertaken only by users with soldering experience. We recommend using a specialised service if you do not feel comfortable with a soldering iron. Installation of the latter also requires soldering experience but that's confined on the *RPi0* board itself and not on the ZX Spectrum Next. On the table below, we list all parts that you will need to perform each upgrade.

| Option | Parts Needed | Notes |
|--------------------|--|--|
| 32K Memory upgrade | 2 x Alliance AS7C34Q98A-10JCNY -01-
2 x Samsung K6M1058V-D-11-0 | Upgrades the memory to 32Kb |
| RTC module | 1 x DS1307 IC
1 x YX3235 32.768KHz 20P oscillator or 32.768KHz
1 x CR1632 Battery holder
1 x CR2032 Battery 3V
1 x 8 pin DIL socket (optional) | Allows time and date keeping that does not only in your computer being powered off |
| WiFi module | ESP8266 ESP-01 | Provides access to the internet and your home network |
| RPi Accelerator | 1 x Raspberry Pi Zero
1 x Female IDC connector 2 x 20 pins | Various functions such as enhanced audio |

Installing a *WiFi* module only requires you to populate the empty socket marked by a **B** on the diagram, page 218, by plugging in the *ESP* module in the place reserved.

Memory is equally simple, however, care must be exercised in that the *RAM* sockets on larger chips have the ones the ZX Spectrum Next has. You need to line up the orientation notch (**B**) of each *RAM* chip (**A**) with the corner of the socket (**D**) leaving space (**C**) in the back of the socket. Once you have everything lined up, push with your finger at the centre of the *RAM* chip and it should make a slight click. While pushing the *RAM* in, and

every other module make sure you provide enough support on the reverse so the board doesn't flex. Refer to the figure below on the proper installation of each RAM chip.

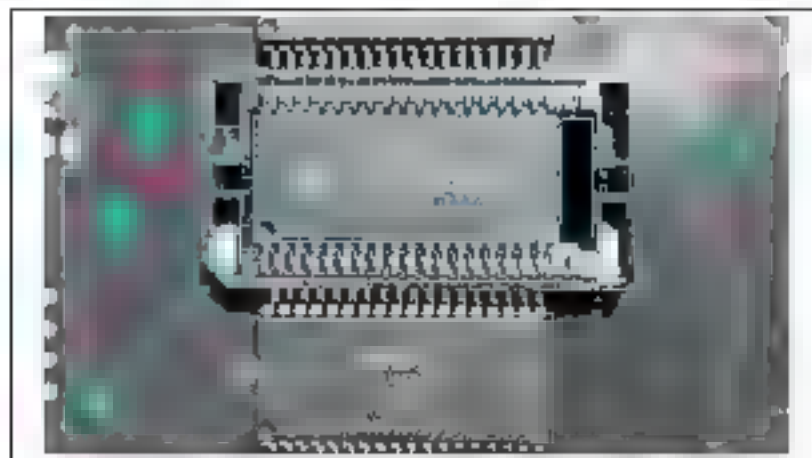


Fig. 34 Optional RAM upgrade installed

The *Raspberry Pi Zero (RPi0)* accelerator requires a little bit of work. You will need to solder the 40 pin (2 x 20) FEMALE IDC header on the RPi0's GPIO through-holes. Unlike what would be normally expected the socket needs to be soldered from the component side, therefore facing downwards. With a properly soldered IDC header you need to be able to see the RPi0's SD card reader and all its components with the IDC header out of view like in the figure below.

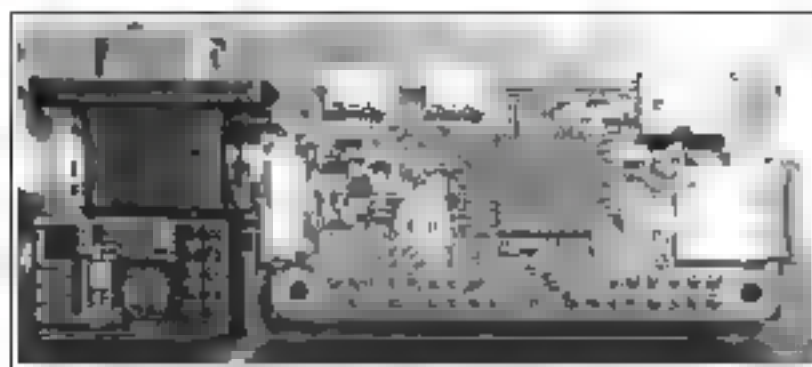


Fig. 35 Raspberry Pi0 installed on Issue 2 board (with WAF module in view - left)

Please note that there is a backplate inside the case covering the holes of where the RPi0's USB Power and Video out will have to appear from. Remove its screws and then pry it out gently with a flat screwdriver before attempting reassembly. Also pay attention to the Quick Start note regarding what is allowed to be plugged in the RPi0. This is also an ideal time to remove the expansion port backplate/cover if you plan on using your ZX Spectrum Next with external interfaces. Figure 36 shows how a ZX Spectrum Next looks disassembled and here's special mention of both backplates.

The most complicated installation is that of the RTC module. It requires you to solder the oscillator in the X₁ location on the board, a battery holder in the location marked and finally the DS1307 IC in its place next to the battery holder paying attention to the orientation (marked by a notch on the sketch on the board as well as on the chip itself). It's advisable that you install a 8 pin DIL socket instead of the DS1307 IC as heat may damage it during soldering.

You should exercise caution while soldering the oscillator, the through holes are very small and need to be free of any flux or solder residue as this will stop the oscillator from working. Finally you will need to install the battery in the socket otherwise the RTC will only work for as long as the machine is powered.

Raspberry Pi Zero installation on the Issue 4 mainboard

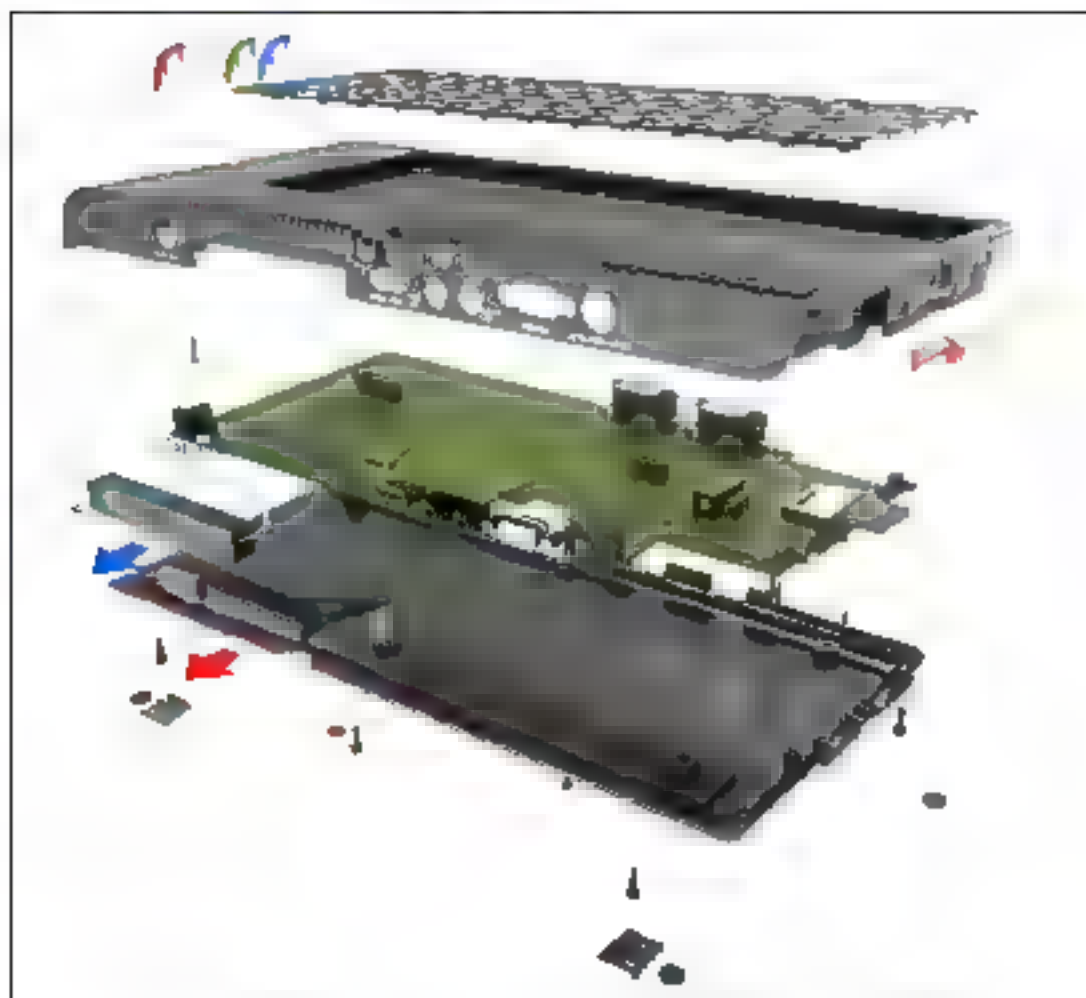


Fig. 36 Disassembled Next. Note the expansion port and Raspberry Pi 0 backplates

The RPi0 installation on the Issue 4 mainboard does not differ in any way from the one done for the Issue 2. So if you don't have an Accelerated Next, follow the instructions in the previous section as they apply here as well.

Notes



The diagram above (Fig. 36) shows a completely disassembled ZX Spectrum Next computer for illustrative purposes only. Not everything can be disassembled: the keyboard itself is considered a standalone unit, fastened into the top frame case by 10 M3 x 4mm screws and YOU SHOULD NOT ATTEMPT TO DISASSEMBLE IT.

USER DISASSEMBLY of the Keyboard unit will NOT be supported by SpecNext Ltd and will invalidate the warranty!

Testing the add-ons installation

Once you have your add-ons installed, it is time to test them. We'll start with the easier tests first, and we'll progress to the most difficult ones.

A Testing the memory

This is by far the simplest test: if your memory installation worked, your *NextZXOS Startup* menu will report 1792K instead of the 768K it reported up until now (see Fig. 37).

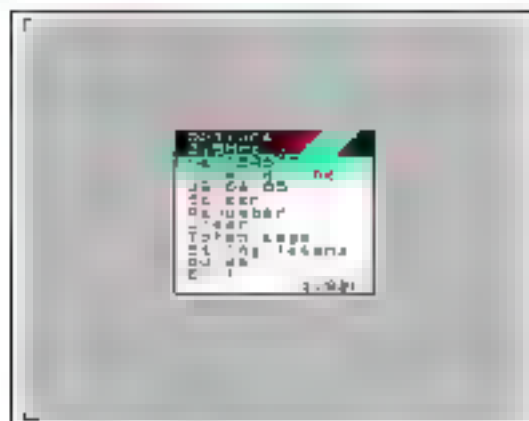


Fig. 3. Net-ZXOS Outputs from simulation /MP

To further verify that the memory was properly installed, here's a program called **2MBTEST (v0.4c).nex** located under **c:/extras/memtest** in your **System/Next™** distribution.

Execute it with the browser or by using the `maxload dot` command and let it go through all your memory testing if's working properly (see Fig. 38 and 39).



Fig. 78 Using the browser to access the Web



40 39 2MB է՝ լուրջ անհույս էր ար

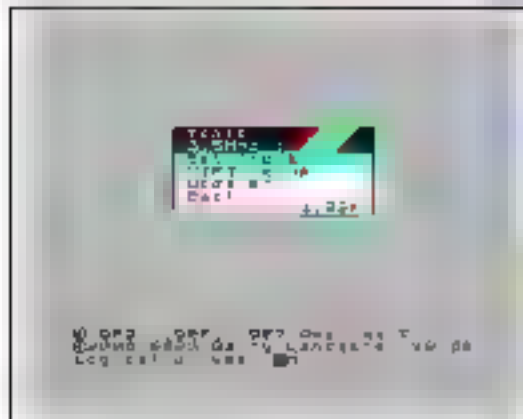
In case that something went wrong, your memory chips are either defective or you didn't install them properly. Make sure your memory chips are properly seated in their sockets by a. checking the space is left as in Fig. 34 and b. pressing them firmly in their sockets until you hear a subtle click sound. If the memory test still fails, your memory chips are probably defective.

B Testing the WiFi

Testing the WiFi feature is a very simple procedure. You only need to use *Next2XOS Startup Menu*, go under *Tools* and then select *WiFi setup*.

As seen in the figures below, you only have to select option 5: Scan Networks, and once your network is found, you enter the password and that was all!

base if it does not work and you have still questions whether or your WiFi module is really problematic you can use the `uart dot` command from your **System/Next™** distribution.



Tip 4. Maximize the value of the objective function.

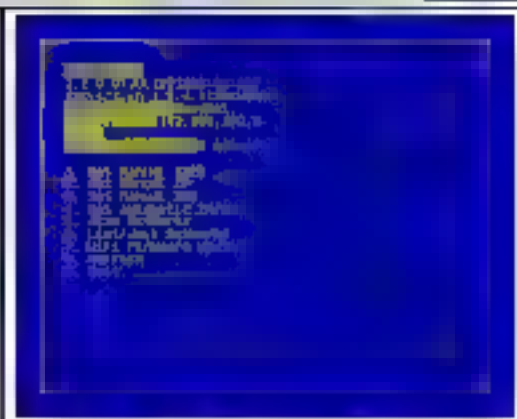


Fig. 4. χ_{eff}^2 versus Δt .

uart is not very complicated but it's quite temperamental especially if you use a PS/2 keyboard. You will need to use the standard ZX Spectrum keys: CAPS SHIFT + O for DELETE SYMBOL SHIFT + K for + SYMBOL SHIFT + G for ? SYMBOL SHIFT + P for ^ SYMBOL SHIFT + N for and SYMBOL SHIFT + L for =

You run it by issuing a

last

you will be greeted by a screen full of information that will end in an `_` cursor. To test type the following.

AT

and press **ENTER**.

If you're good so far, the ESP will be responding with

○

That's a very good sign. That means serial communications have been established. To see however if the ESP is actually working, you'll need to issue a few more commands. Type

AT 4CJMODE?

The ESP here should respond with a 1, 2 or 3 (this is the mode that's its working at: being 1 for Station, 2 for Access Point and 3 for both). Normally this should be enough to verify your ESP is working but if you want to take it one step further you should set the ESP to station mode by giving:

AT +CWMODE=1

then check for what Access Points are around by doing

AT+CU,AP

before finally connecting to `ord` by giving the command:

```
AT+CUJAP="SSID ", 'YourPass '
```

where SSID is the name of your network and YourPass is your WiFi password. The ESP will retain these so you can do it you want!

 $AT + RS^T$

which will reset your ESP and give you a lot of information before concluding with a

WIFI CONNECTED
WIFI GOT IP

Exit **uart** by pressing **SYMBOL SHIFT + SPACE**. If none of this worked, then the most likely culprits are that you either have a bad ESP module or that the power supply you're using is not powerful enough for both your ZX Spectrum Next and the ESP module. First try with a different power supply, otherwise return the ESP module for an exchange.

Note that different ESP firmware versions have slightly different versions of the commands above so always consult the most up-to-date documentation.

C Testing the RTC installation

There are several ways of testing if the RTC was properly installed, the easiest of which is to launch again the *NextZXOS Startup Menu* and then go to the *Tools Submenu*. There apart from the *WiFi setup* option, you will find a *Set Clock* option. Launching it will allow you to program your RTC using just your cursor and the **ENTER** keys.

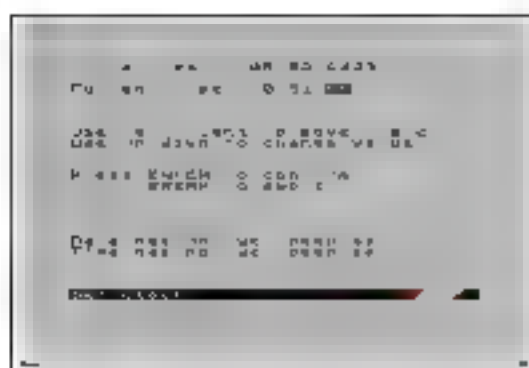


Fig. 42 NextZXOS Set Clock utility

If everything worked right, then *NextZXOS* shall start reporting the current time and date on its menus like so:

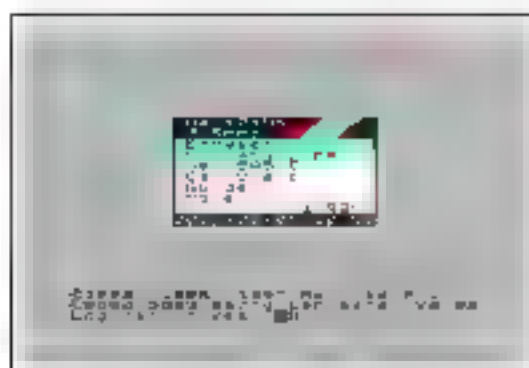


Fig. 43 RTC Working

If however the clock won't work, there's a very simple way of testing for the RTC and that's to give it the *time* command. If it doesn't work out right, it will produce an output like the one shown in Fig. 44.

There are a few issues that can occur with the RTC. If the output is as above, the most likely culprit is the soldering of the IC onto the board. A cold solder will leave the RTC not working, a short somewhere will do the same, but the RTC IC will start getting hot. If you feel the IC warming up, disconnect all power immediately and inspect your soldering.

A defective battery hinder installation as well as a defective or depleted battery will manifest itself with the R/C not keeping the user boot-up and NextZXOS not displaying the time and date information on its Startup Menu. Setting the time and date anew, however, will restore the time display.



Fig. 44 R/C not working

On the other hand the commands described in the *Using the Real Time Clock hardware* section below do work: the time and date information appear in the Startup menu but time does not advance: then the issue lays usually with either the oscillator or the pins of the LM-327 IC these connect to: here's either a short somewhere or even a situation as simple as leftover flux from soldering. The oscillator can be damaged quite easily so make sure there's no continuity on its two legs before even turning the power on and inserting the battery in the holder.

D Testing the Accelerator installation

Before you can test that the accelerator is working, there are a few things you need to do. First, find a 16 GiB or larger microSD-Card and then you need to download the NextPi2 distribution from <http://zxkallor.com/NextPi2> together with the instructions that accompany it.

Once you prepare the SD card according to the instructions, put it in your Pi Accelerator prior to booting up your ZX Spectrum Next. The Pi support programs are already in the `c:\apps\pi\folder` or your `System\Next` distribution's `SL`, so you do not need to do anything else other than powering up the machine.

When you have access to the RPi0 you should see the green led flashing while the ZX Spectrum Next is booting: that's a good first sign showing that the RPi0 is loading its NextPi2 distribution. The LED will eventually stop flashing and should turn into a steady green. Once you're all booted up, change to the NextPi2 support folder, switch in the `terminex` folder and execute it, in the browser or with the `SPECTRUM` command `terminex.snx` by David Taphie. If the RPi0 installation worked, you will see a message stating `Connection to NextPi established` followed by a `SUP>` prompt which means your RPi0 installation was successful as shown in the figure below.



Fig. 45 RPi Supervisor prompt via Terminex

If the **SUP>** prompt does not appear after a maximum of 20-25 seconds, that means there's something wrong. That doesn't mean your RPi0 is not working, especially if you saw the flashing green LED light on it earlier. This more than likely means that you didn't transfer the NextPI2 image properly or that there's some problem with the microSD card you used.

To verify the RPi0 is working, you will need to unplug it from your ZX Spectrum Next, locate a micro-usb power supply, an appropriate HDMI cable and a standard RPi0 distribution and power it independently.

If you can see output on the screen, then there's either a problem with your NextPI2 SD card, which you can verify by plugging its microSD card in the RPi0's reader instead of the standard RPi0 distribution, a cold solder on your ID17 connector (or you soldered earlier), or finally, an insufficiently powerful, power supply for your ZX Spectrum Next.

The RPi0s are very resilient pieces of hardware and they don't fail easily. Chances are any failure you experience is due to one of the cases listed.

Using the Real Time Clock hardware

If you're lucky to have an expanded ZX Spectrum Next with the battery backed-up Real Time Clock (RTC) hardware installed, or if you followed the instructions to install it yourself, then more options in timekeeping become available to you. These options do not suffer from the drawbacks and caveats laid out in the previous sections as this dedicated hardware option keeps time regardless of what else the computer is doing and in fact keeps time even when the computer is turned off.

The RTC is accessible via the function **T.MES** (See Chapter 17) as well as two commands: **date** and **time**.

Setting up your RTC for first use

As we saw above, invoking the Set Clock option found in the Tools submenu of the NextZXOS Startup Menu has the obvious benefit of setting up the clock AND testing at the same time. There are however two more ways to set your RTC up, especially if for some reason you wish to use an alternative to NextZXOS like for example esxZXS.

Using time and date

This is very straightforward with the small exception that before you can use **date** and **time** you will need to set up your Real Time Clock hardware. Luckily this is only done once when you install it and whenever you need to change battery. You will initially (for safety) need to issue the command

```
time di
```

This wipes the RTC signature from the chip and gets it ready to accept a date and time. You can then type

```
time '10:35:23'
```

where '10:35:23' can be substituted by any string of the format HH:MM:SS where HH (hour) is a number from 00 to 23, MM (minute) is a number from 00 to 59 and SS (seconds) is a number from 00 to 59. You then enter the correct date by issuing

```
.date '20/03/2023'
```

where '20/03/2023' can be substituted by any string of the format DD/MM/YYYY where DD is the day (01 to 31), MM is the month (01 to 12) and YYYY is any year from 2000 to 2099.

A few interesting things will happen once you install and setup your RTC. First, NextZXOS will report the time and date on its Startup menu (which is very nice indeed). Then, your saved files will start having a date and time stamp to them (visible with **CAT EXP** or **ls**).

Using the RTC together with the WiF module

The RTC module is not very accurate and can lose several seconds over the period of a few weeks. Luckily like other much larger systems, the ZX Spectrum Next can also set its time from the internet thanks to `nntp`, the dot command client to Robin Verhagen-Guest's *Next Time Protocol* server. Its syntax is

```
nntp server-address port [-z=Timezone]
```

where *server-address* is a FQDN or IP address running a `nntp` server, *port* is the port where that `nntp` server is listening to (by default 12300) and an optional *timezone* parameter to set the time to any location you would like from a list of acceptable timezones

```
nntp time zx.in.net 12300 -z=UTC
```

will talk to the the `nntp` server located at time `zx.in.net`, listening on port 12300 and set the RTC's time to **Coordinated Universal Time (UTC)** whereas

```
.nntp time .zx.in.net 12300 z=GMT
```

will do the same but for **Greenwich Mean Time** meaning the time will adjust for summer giving you **BST** and winter giving you **UTC** as `nntp` already knows about *daylight savings*. It will work this into your RTC's time setting meaning you never have to worry about setting your clock in the summer or winter provided your location observes these

A full list of accepted timezones exists at the `nntp` project's wik page located at <https://github.com/Threetwosevenseven/nntp/wiki/Timezone-Codes>

It is a good idea if you have an always working WiF setup to add `nntp` to your **autoexec.bas** file so it always sets the correct time whenever your ZX Spectrum Next boots. The potential start up delay is very small and the benefit of always having correct time outweighs the delay

Using the rest of the add-ons

Both the WiF and Raspberry Pi Accelerator add-ons open up exciting features not before seen in a ZX Spectrum computer. This chapter provides only limited coverage as the feature set of both is still evolving. We have included all features implemented thus far, Audio playback, TZX loading, DRIVEA support etc. in Chapters 18, 19, 21 and in this chapter however you're encouraged to read the accompanying documentation found in your **System/Next™** distribution and on www.specnext.com as they will always contain the most up-to-date information regarding these add-ons and newer ZX Spectrum Next features

Chapter 22 IN, OUT and the Next Registers

We can instruct the processor to read from and write to memory by using **PEEK** **POKE** and other variants. For all the possibilities, see the chapter **The Memory**. The processor itself does not really care whether memory is RAM, ROM or even nothing at all; it just knows that there are **65536** memory addresses, and it can read a byte from each one. Even the **6502** has an **IN** and **OUT** instruction, but it never really gets used, because the processor is steadily in a continuously changing way, there are also **65536** hardware addresses, called **I/O ports**, input/output ports, that are separate from memory. These are used by the processor for communicating with attached devices like the keyboard or the display, and they can be controlled from **NextZ80** by using the **IN** function and the **OUT** statement. There's also a **PEEK** and **POKE** statement for the **NextZ80**. **NextZ80** has a few other varied functions, they too are accessible with **IN** and **OUT**, but two of them are also accessible via a special dual statement/function, called **REG**.

IN and OUT

IN is a function like the simplest form of **PEEK**.

IN port

has the form **IN port**, where the hardware address **port** and its result is a byte, same as **PEEK**. **OUT** on the other hand is a statement like a simple **POKE**.

OUT port v

which writes value **v** to the hardware address **port**.

Hardware address decoding

How the address is interpreted depends on the hardware of the computer and attached devices. In previous versions of the **ZX Spectrum**, there were several and essentially illegal peripheral, many different port addresses mapped to the same device. This is called **port conflict** and happened because some address bits were ignored in the hardware to save costs. As a consequence of the changes in the **NextZ80** we preserve this by the virtual peripherals. This made it hard to new peripherals, both for non-conflicting ports to use and to make a variety of input and output ports that didn't conflict with the most popular peripherals. The situation was somewhat mitigated by the fact that only a couple of peripherals could be connected to the older **ZX Spectrum** machines at a time, the **elementary** peripherals, where the modern **NextZ80** implementation allows many devices in the hardware. This port conflict problem returns with renewed vigour, as the parallel bus is now virtually any port address and it is fun to make a port conflict.

The **ZX Spectrum Next** fully decodes port addresses for new peripherals, meaning it does not generally address the **NextZ80** as a whole, but has two different bus addresses for each device. These must continue to be manually decoded, but the developer has to understand the issues a hard and in the table that follows which contains all available port addresses in the **ZX Spectrum Next**. So far, as we approach the **NextZ80** as writer, in any way we can easily show which bits are being ignored by a specific peripheral. Each hardware address is 16 bits wide, which we shall call (using **A** for address)

| | | | | | | | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|----|----|----|----|----|----|----|----|----|----|
| A15 | A14 | A13 | A12 | A11 | A10 | A9 | A8 | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 |
|-----|-----|-----|-----|-----|-----|----|----|----|----|----|----|----|----|----|----|

where **A0** is the 1st bit, **A1** the 2nd bit, **A2** the 3rd bit, **A3** the 4th bit, and so on. The table that follows shows which bits are ignored by the **NextZ80** and which means we respond in all 32768 even port addresses, a 16 bit address, that is, from **254 FEh**. The byte value which has 8 bits, and these are often referred to (using **D** for data) as

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|----|----|----|----|----|----|----|----|

Here is a list of the port addresses used with next decoding. For the reason mentioned, only the JLA has an even port address and every even-numbered port ILN will result in the JLA being read.

| R | W | A15 | A14 | A13 | A12 | A11 | A10 | A9 | A8 | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 | Port (Hex) | Description | |
|---|---|-----|-----|-----|-----|-----|-----|----|----|----|----|----|----|----|----|----|----|------------|-------------------------------------|-------------------------------------|
| ■ | ■ | | | | | | | | | | | | | | | | | 1h | JLA | |
| ■ | ■ | | | | | | | | | 1 | 1 | 1 | 1 | | | | | FFh | Timex video Floating bus | |
| ■ | ■ | 0 | | | | | | | | | | | | | | | 0 | 7FFh | Memory Paging Control | |
| ■ | ■ | | | | | | | | | | | | | | | | 1 | 7FFh | Memory Paging Control (3/Next only) | |
| ■ | ■ | 1 | 1 | 0 | † | | | | | | | | | | | | 0 | 0FFh | Next Memory Bank Select† | |
| ■ | ■ | 0 | 0 | 0 | | | | | | | | | | | | | 0 | 1 | FFh | 3 Memory Paging Control |
| ■ | ■ | | | | | | | | | | | | | | | | | 2 | FFh | 3 DC Status |
| ■ | ■ | 0 | 0 | | | | | | | | | | | | | | | 3 | FFh | 3 Memory Control |
| ■ | ■ | | | | | | | | | | 1 | | | | | | | EF | 7h | Pentagon 024K Memory Paging Control |
| ■ | ■ | 0 | 0 | 0 | 0 | | | | | | | | | | | | 0 | | | 3 Floating bus |
| ■ | ■ | 0 | 0 | | | | | 0 | 0 | 0 | 0 | | | | | | | 24 | 3Bh | NextHLS Select |
| ■ | ■ | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 25 | 3Bh | NextREG Data |
| ■ | ■ | 1 | 1 | | | | | 0 | 0 | 0 | | | | | | | | 26 | 3Bh | PT SCL |
| ■ | ■ | 0 | 0 | | | | | 0 | | 0 | | | | | | | | 1B | 3Bh | PT SCL |
| ■ | ■ | 0 | | | | | | 1 | 0 | 0 | | | | | | | | 24 | 3Bh | PT SCL |
| ■ | ■ | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 37 | 3Bh | JART Tx |
| ■ | ■ | | | | | | | 0 | 1 | | | | | | | | | 42 | 3Bh | JART Rx |
| ■ | ■ | 0 | 0 | | | | | 0 | 0 | 0 | | | | | | | | 53 | 3Bh | JART select |
| ■ | ■ | | | 1 | 1 | 1 | 1 | | | | | | | | | | 1 | 63 | 3Bh | JART Frame |
| ■ | ■ | 0 | 0 | | | | | 0 | 0 | | | | | | | | | 42 | 3Bh | PT SCL |
| ■ | ■ | 0 | 0 | | | | | 0 | 0 | | | | | | | | | 93 | 3Bh | PT SCL Channel 1 |
| ■ | ■ | 0 | | | | | | 0 | 0 | | | | | | | | | 43 | 3Bh | PT SCL Channel 2 |
| ■ | ■ | 0 | 0 | | | | | 0 | 0 | | | | | | | | | 93 | 3Bh | PT SCL Channel 3 |
| ■ | ■ | 0 | 0 | 0 | † | | | 0 | 0 | 1 | 1 | | 0 | 1 | 1 | | | 103 | 3Bh | PT SCL Channel 4† |
| ■ | ■ | 0 | 0 | | | | | 0 | 0 | | | | | | | | | 00 | 3Bh | PT SCL Channel 5† |
| ■ | ■ | | 0 | | | 1 | | | | | | | | | | 1 | 1 | 43 | 3Bh | PT SCL Channel 6† |
| ■ | ■ | 0 | 0 | | | | | 0 | 0 | | | | | | | | | 00 | 3Bh | PT SCL Channel 7† |
| ■ | ■ | | 0 | | | | | | 0 | | | | | | | | | 93 | 3Bh | PT SCL Channel 8† |
| ■ | ■ | 1 | 1 | 1 | † | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | | FF | 3Bh | JLAplus Data |
| ■ | ■ | | | | | | | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | | | 0Bh | | zBOUMA† |
| ■ | ■ | | | | | | | | | | | | | | | | | FFh | | zBOUMA |
| ■ | ■ | | | | | | | | | | | | | | | | | FFh | | zBOUMA |
| ■ | ■ | | 0 | | | | | | | | | | | | | | 1 | BFFh | AY Data (readable on 3/Next only) | |
| ■ | ■ | 1 | 0 | | | | | | | | | | | 0 | 1 | 0 | 1 | BFFh | AY int (inside BFFh decoding) | |
| ■ | ■ | | | | | | | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | | | FFh | | DAC A |
| ■ | ■ | | | | | | | | | | | | | | | | | FFh | | DAC A |
| ■ | ■ | | | | | | | 0 | 0 | 1 | 1 | | 1 | 1 | | | | 3Fh | | DAC A |
| ■ | ■ | | | | | | | 0 | 0 | 0 | 0 | | 1 | 1 | 1 | | | 0Fh | | DAC B |
| ■ | ■ | | | | | | | 1 | | 1 | 1 | | | | | | | 3Fh | | DAC B |
| ■ | ■ | | | | | | | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | | | 0Fh | | DAC A.D |
| ■ | ■ | | | | | | | 1 | | | | | | | | | | FFh | | DAC A.D |
| ■ | ■ | | | | | | | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | | | B3h | | DAC B.C |
| ■ | ■ | | | | | | | 0 | 1 | 0 | 0 | | | | | | | 4Fh | | DAC C |
| ■ | ■ | | | | | | | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | | | FFh | | DAC C |
| ■ | ■ | | | | | | | 0 | 1 | 0 | 1 | | | | | | | 5Fh | | DAC D |
| ■ | ■ | | | | | | | 1 | | 0 | 0 | | | | | | | FFh | | DAC D |
| ■ | ■ | | | | | | | 1 | 1 | 0 | | | | 1 | 1 | | | EBh | | SPI C.A.T.A |
| ■ | ■ | | | | | | | 1 | 1 | 0 | | | | | | | | FFh | | SPI C.A.T.A |

† Readable only on AY

‡ Readable only on AY (readable on 3/Next only)

1 Subsequent bits (00000000) are not used in the DMA program

2 The sequence over on L

| R | W | A15 | A14 | A13 | A12 | A11 | A10 | A9 | A8 | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 | Port (Hex) | Description |
|---|---|-----|-----|-----|-----|-----|-----|----|----|----|----|----|----|----|----|----|----|------------|------------------------------|
| ■ | | | | | | 1 | | | 1 | | 0 | 1 | | | | | | F4DFh | KEMPSTON Mouse x |
| ■ | | | | | | | | | | 1 | 0 | 1 | | | | | | FFDFh | KEMPSTON Mouse Y |
| ■ | | | | | | 1 | 1 | 1 | | 1 | 1 | 1 | | | | | | F4DFh | KEMPSTON Mouse Wheel Buttons |
| ■ | | | | | | | | | | 0 | 0 | 0 | 1 | | | | | Fh | KEMPSTON Joystick 1 |
| ■ | | | | | | | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | DFh | KEMPSTON Joystick Alias |
| ■ | | | | | | | | | | 0 | 0 | | 1 | 0 | | | | 37h | KEMPSTON Joystick 2 |
| ■ | ■ | | | | | | | | | 0 | 0 | 0 | 1 | | | | 1 | Fh | Multiface 1 Disable |
| ■ | ■ | | | | | | | | | 1 | 0 | 0 | 1 | | | | | 9Fh | Multiface 1 Enable |
| ■ | ■ | | | | | | | | | 0 | 0 | 0 | 1 | | | | | Fh | Multiface 128 v87 12 Disable |
| ■ | ■ | | | | | | | | | 1 | 0 | 0 | 1 | | | | | 9Fh | Multiface 128 v87 12 Enable |
| ■ | ■ | | | | | | | | | 0 | 0 | | 1 | | | | | 3Fh | Multiface 128 v87 2 Disable |
| ■ | ■ | | | | | | | | | 1 | 0 | | 1 | | | 1 | 1 | BFh | Multiface 128 v87 2 Enable |
| ■ | ■ | | | | | | | | | 1 | 0 | | 1 | | | | | BFh | Multiface +3 Disable |
| ■ | ■ | | | | | | | | | 1 | 0 | | 1 | | | | | 3Fh | Multiface +3 Enable |
| ■ | ■ | 0 | | | | | | 1 | 1 | 0 | 0 | | 1 | | | | | 303Bh | Sprite slot flags |
| | ■ | | | | | | | | | 0 | 1 | 0 | 1 | 0 | | | | 57h | Sprite Attributes |
| | ■ | | | | | | | | | 1 | 0 | 1 | | 0 | | | | 5Bh | Sprite Pattern |

The above table describes the ports that are used to control and communicate with additional hardware features on the ZX Spectrum Next. There are also 17 ports that are of special significance as they are unique to the ZX Spectrum Next. These are:

| Port Name | Address (Hex) | Address (Dec) | Description |
|----------------|---------------|---------------|--|
| NextReg Select | 243Bh | 955F | Communicates with the ZX Spectrum Next hardware |
| NextReg Data | 253Bh | 965F | Used after register selection to send and read data |
| PC_SCL | 03Bh | 40B6 | Used for PC device communication (I2C, etc) |
| PC_SDA | 3Bh | 41 | Sends/Receives data from/to the PC bus |
| Layer 2 | 29Bh | 40E | Used to control Layer 2 |
| UART Tx | 33Bh | 4972 | Transmits data from the UART |
| UART Rx | 43Bh | 5179 | Receives data from the UART |
| UART Select | 53Bh | 5436 | Selects which UART is in use |
| UART Frame | 63Bh | 566 | Sets up the UART framing |
| CTC | 43Bh - 1FB3h | 5203 - 8115 | Configures the Counter-Timer circuits |
| z80DMA | 0Bh | 11 | Programs the DMA in z80DMA compatible mode |
| zx0DMA | 6Bh | 107 | Programs the DMA in zx0DMA mode |
| SPI Select | E7h | 227 | Selects an SPI peripheral (SD Card/Flash/ROM/RFID) |
| SPI DATA | EBh | 235 | Sends/Receives Data via the SPI bus a byte at a time |
| SPRITE | 303Bh | 244 | Controls the Next Sprite Engine |
| SPRITE ATTR | 57h | 8 | Sends Sprite Attributes to the Sprite Engine |
| SPRITE PATTERN | 5Bh | 91 | Sends Sprite Pattern to the Sprite Engine |

Two of the most important ports in this I/O are collectively called *Next Register* or *NextREG* for short. Most of the machine's features can be controlled through *NextREG*.

Accessing the ZX Spectrum Next features with NextREG

We use a *NextREG* by first selecting it with the control port and then writing to or reading from the *NextREG* data port.

In *NextBASIC* this is achieved with two consecutive **OUT** commands in the case of writing or with a combination of consecutive **OUT** and **IN** in the case of reading from a *NextREG*.

The first command is directed to the Select port **9276** (**243Bh**) selecting a specific register and the second to the Data port **9531** (**2538h**) to modify or read the value stored there.

Given from *NextBASIC* these commands must be given consecutively in one line as *NextBASIC* may do something different with *NextREG* in-between commands. If you give the first and then wait to give the second *NextBASIC* may have changed the Select register in the meantime, so by giving them together you give it no time to do something else.

The Z80N CPU which powers the ZX Spectrum Next also provides a special **NEXTREG** instruction and this is referenced in *Appendix A*.

Finally, as mentioned in the introduction, in order to read *NextREG*, *NextBASIC* also has a specialised command and function called **REG**, which is much easier to use than the combination of **OUT** and **IN** keywords.

We'll showcase both methods here in order for you to be able to use either as there are cases where the ZX Spectrum Next's facilities are still available but without *NextBASIC* (it provides access to them). The command as a statement has the form

```
REG n,v
```

which is essentially the same as doing

```
OUT 9275, n OUT 9531, v
```

Obviously *n* is the register number and *v* is the value we modify the register with. As a function **REG** has the following form

```
% REG n or REG n
```

as it works with both the integer as well as the standard expression evaluators. This essentially is the same as executing

```
OUT 9275, n: %x = % IN 9531 (or the equivalent x = IN 9531)
```

Let's give one simple example in both forms and let's mix-and-match a bit as well, to show the equivalency.

Assuming we want to change speeds to **28MHz** we could give,

```
RUN AT 3  
  
OUT 9275, 7 OUT 9531, 3
```

and we can verify that it is set by either bringing up any *NextBASIC* menu, menu's list the currently set speed on top) with the EDIT key or by doing,

```
OUT 9275,7 PRINT % IN 9531 & @11
```

Which is the same as

```
PRINT % REG 7&@11
```


You can verify this actually changes things by doing a **RUN AT 2** and give the **OUTIN** sequence again. As you will see from the list that follows, not every *NextREG* is dedicated solely to one function. In this case the only bits that concerned us were Bits 0 and 1 and that's why we used a 2-bit bitmask with the bitwise **AND** operator **&**. For the same example using just the **REG** command, our line would have been as simple as

```
REG 7 3
```

In our example in Chapter 17 where we read *NextREG* 5n we also used bit shifting which is a great way to get the value of a single bit in a register. In our case we only needed bit 2 of the register so after getting the specific bit by bitwise **AND** **&** the register value with a 3-bit bitmask, we shifted it two places 'bits' to the right by using the right bit-shifting operator **{>>}**. That way we were able to get the value of the single register bit.

Generally speaking, you will often want to modify individual bits in a *NextREG* without changing the remaining ones.

You can do this by first reading the *NextREG* and then *masking off* the bits you want to leave unchanged by using bitwise **AND** **&**, and finally write that value with the new bits added in.

We'll list all of *NextREG*s below in numerical order. Not every register is accessible, so pay attention to the key at the start of the list to understand whether a register can be read, written or both.

[illegible][illegible]

Macro 2: applying statistical tests

Figure 1. Schematic diagram of the proposed system. The system consists of a user, a server, and a database. The user sends a request to the server, which then queries the database. The database returns the results to the server, which then sends the response back to the user.

| Year | 1990 | 1991 | 1992 | 1993 | 1994 | 1995 | 1996 | 1997 | 1998 | 1999 | 2000 | 2001 | 2002 | 2003 | 2004 | 2005 | 2006 | 2007 | 2008 | 2009 | 2010 | 2011 | 2012 | 2013 | 2014 | 2015 | 2016 | 2017 | 2018 | 2019 | 2020 | 2021 | 2022 | 2023 | 2024 | 2025 | 2026 | 2027 | 2028 | 2029 | 2030 | 2031 | 2032 | 2033 | 2034 | 2035 | 2036 | 2037 | 2038 | 2039 | 2040 | 2041 | 2042 | 2043 | 2044 | 2045 | 2046 | 2047 | 2048 | 2049 | 2050 | 2051 | 2052 | 2053 | 2054 | 2055 | 2056 | 2057 | 2058 | 2059 | 2060 | 2061 | 2062 | 2063 | 2064 | 2065 | 2066 | 2067 | 2068 | 2069 | 2070 | 2071 | 2072 | 2073 | 2074 | 2075 | 2076 | 2077 | 2078 | 2079 | 2080 | 2081 | 2082 | 2083 | 2084 | 2085 | 2086 | 2087 | 2088 | 2089 | 2090 | 2091 | 2092 | 2093 | 2094 | 2095 | 2096 | 2097 | 2098 | 2099 | 2100 |
|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| Year | 1990 | 1991 | 1992 | 1993 | 1994 | 1995 | 1996 | 1997 | 1998 | 1999 | 2000 | 2001 | 2002 | 2003 | 2004 | 2005 | 2006 | 2007 | 2008 | 2009 | 2010 | 2011 | 2012 | 2013 | 2014 | 2015 | 2016 | 2017 | 2018 | 2019 | 2020 | 2021 | 2022 | 2023 | 2024 | 2025 | 2026 | 2027 | 2028 | 2029 | 2030 | 2031 | 2032 | 2033 | 2034 | 2035 | 2036 | 2037 | 2038 | 2039 | 2040 | 2041 | 2042 | 2043 | 2044 | 2045 | 2046 | 2047 | 2048 | 2049 | 2050 | 2051 | 2052 | 2053 | 2054 | 2055 | 2056 | 2057 | 2058 | 2059 | 2060 | 2061 | 2062 | 2063 | 2064 | 2065 | 2066 | 2067 | 2068 | 2069 | 2070 | 2071 | 2072 | 2073 | 2074 | 2075 | 2076 | 2077 | 2078 | 2079 | 2080 | 2081 | 2082 | 2083 | 2084 | 2085 | 2086 | 2087 | 2088 | 2089 | 2090 | 2091 | 2092 | 2093 | 2094 | 2095 | 2096 | 2097 | 2098 | 2099 | 2100 |

[illegible]

1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 13. 14. 15. 16. 17. 18. 19. 20. 21. 22. 23. 24. 25. 26. 27. 28. 29. 30. 31. 32. 33. 34. 35. 36. 37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50. 51. 52. 53. 54. 55. 56. 57. 58. 59. 60. 61. 62. 63. 64. 65. 66. 67. 68. 69. 70. 71. 72. 73. 74. 75. 76. 77. 78. 79. 80. 81. 82. 83. 84. 85. 86. 87. 88. 89. 90. 91. 92. 93. 94. 95. 96. 97. 98. 99. 100. 101. 102. 103. 104. 105. 106. 107. 108. 109. 110. 111. 112. 113. 114. 115. 116. 117. 118. 119. 120. 121. 122. 123. 124. 125. 126. 127. 128. 129. 130. 131. 132. 133. 134. 135. 136. 137. 138. 139. 140. 141. 142. 143. 144. 145. 146. 147. 148. 149. 150. 151. 152. 153. 154. 155. 156. 157. 158. 159. 160. 161. 162. 163. 164. 165. 166. 167. 168. 169. 170. 171. 172. 173. 174. 175. 176. 177. 178. 179. 180. 181. 182. 183. 184. 185. 186. 187. 188. 189. 190. 191. 192. 193. 194. 195. 196. 197. 198. 199. 200. 201. 202. 203. 204. 205. 206. 207. 208. 209. 210. 211. 212. 213. 214. 215. 216. 217. 218. 219. 220. 221. 222. 223. 224. 225. 226. 227. 228. 229. 230. 231. 232. 233. 234. 235. 236. 237. 238. 239. 240. 241. 242. 243. 244. 245. 246. 247. 248. 249. 250. 251. 252. 253. 254. 255. 256. 257. 258. 259. 260. 261. 262. 263. 264. 265. 266. 267. 268. 269. 270. 271. 272. 273. 274. 275. 276. 277. 278. 279. 280. 281. 282. 283. 284. 285. 286. 287. 288. 289. 290. 291. 292. 293. 294. 295. 296. 297. 298. 299. 300. 301. 302. 303. 304. 305. 306. 307. 308. 309. 310. 311. 312. 313. 314. 315. 316. 317. 318. 319. 320. 321. 322. 323. 324. 325. 326. 327. 328. 329. 330. 331. 332. 333. 334. 335. 336. 337. 338. 339. 340. 341. 342. 343. 344. 345. 346. 347. 348. 349. 350. 351. 352. 353. 354. 355. 356. 357. 358. 359. 360. 361. 362. 363. 364. 365. 366. 367. 368. 369. 370. 371. 372. 373. 374. 375. 376. 377. 378. 379. 380. 381. 382. 383. 384. 385. 386. 387. 388. 389. 390. 391. 392. 393. 394. 395. 396. 397. 398. 399. 400. 401. 402. 403. 404. 405. 406. 407. 408. 409. 410. 411. 412. 413. 414. 415. 416. 417. 418. 419. 420. 421. 422. 423. 424. 425. 426. 427. 428. 429. 430. 431. 432. 433. 434. 435. 436. 437. 438. 439. 440. 441. 442. 443. 444. 445. 446. 447. 448. 449. 450. 451. 452. 453. 454. 455. 456. 457. 458. 459. 460. 461. 462. 463. 464. 465. 466. 467. 468. 469. 470. 471. 472. 473. 474. 475. 476. 477. 478. 479. 480. 481. 482. 483. 484. 485. 486. 487. 488. 489. 490. 491. 492. 493. 494. 495. 496. 497. 498. 499. 500. 501. 502. 503. 504. 505. 506. 507. 508. 509. 510. 511. 512. 513. 514. 515. 516. 517. 518. 519. 520. 521. 522. 523. 524. 525. 526. 527. 528. 529. 530. 531. 532. 533. 534. 535. 536. 537. 538. 539. 540. 541. 542. 543. 544. 545. 546. 547. 548. 549. 550. 551. 552. 553. 554. 555. 556. 557. 558. 559. 560. 561. 562. 563. 564. 565. 566. 567. 568. 569. 570. 571. 572. 573. 574. 575. 576. 577. 578. 579. 580. 581. 582. 583. 584. 585. 586. 587. 588. 589. 590. 591. 592. 593. 594. 595. 596. 597. 598. 599. 600. 601. 602. 603. 604. 605. 606. 607. 608. 609. 610. 611. 612. 613. 614. 615. 616. 617. 618. 619. 620. 621. 622. 623. 624. 625. 626. 627. 628. 629. 630. 631. 632. 633. 634. 635. 636. 637. 638. 639. 640. 641. 642. 643. 644. 645. 646. 647. 648. 649. 650. 651. 652. 653. 654. 655. 656. 657. 658. 659. 660. 661. 662. 663. 664. 665. 666. 667. 668. 669. 670. 671. 672. 673. 674. 675. 676. 677. 678. 679. 680. 681. 682. 683. 684. 685. 686. 687. 688. 689. 690. 691. 692. 693. 694. 695. 696. 697. 698. 699. 700. 701. 702. 703. 704. 705. 706. 707. 708. 709. 710. 711. 712. 713. 714. 715. 716. 717. 718. 719. 720. 721. 722. 723. 724. 725. 726. 727. 728. 729. 730. 731. 732. 733. 734. 735. 736. 737. 738. 739. 740. 741. 742. 743. 744. 745. 746. 747. 748. 749. 750. 751. 752. 753. 754. 755. 756. 757. 758. 759. 760. 761. 762. 763. 764. 765. 766. 767. 768. 769. 770. 771. 772. 773. 774. 775. 776. 777. 778. 779. 780. 781. 782. 783. 784. 785. 786. 787. 788. 789. 790. 791. 792. 793. 794. 795. 796. 797. 798. 799. 800. 801. 802. 803. 804. 805. 806. 807. 808. 809. 810. 811. 812. 813. 814. 815. 816. 817. 818. 819. 820. 821. 822. 823. 824. 825. 826. 827. 828. 829. 830. 831. 832. 833. 834. 835. 836. 837. 838. 839. 840.

1. **NAME** _____
 2. **DATE** _____
 3. **TIME** _____
 4. **LOCATION** _____
 5. **REASON** _____
 6. **REMARKS** _____
 7. **SIGNATURE** _____
 8. **DATE** _____
 9. **TIME** _____
 10. **LOCATION** _____
 11. **REASON** _____
 12. **REMARKS** _____
 13. **SIGNATURE** _____
 14. **DATE** _____
 15. **TIME** _____
 16. **LOCATION** _____
 17. **REASON** _____
 18. **REMARKS** _____
 19. **SIGNATURE** _____
 20. **DATE** _____
 21. **TIME** _____
 22. **LOCATION** _____
 23. **REASON** _____
 24. **REMARKS** _____
 25. **SIGNATURE** _____
 26. **DATE** _____
 27. **TIME** _____
 28. **LOCATION** _____
 29. **REASON** _____
 30. **REMARKS** _____
 31. **SIGNATURE** _____
 32. **DATE** _____
 33. **TIME** _____
 34. **LOCATION** _____
 35. **REASON** _____
 36. **REMARKS** _____
 37. **SIGNATURE** _____
 38. **DATE** _____
 39. **TIME** _____
 40. **LOCATION** _____
 41. **REASON** _____
 42. **REMARKS** _____
 43. **SIGNATURE** _____
 44. **DATE** _____
 45. **TIME** _____
 46. **LOCATION** _____
 47. **REASON** _____
 48. **REMARKS** _____
 49. **SIGNATURE** _____
 50. **DATE** _____
 51. **TIME** _____
 52. **LOCATION** _____
 53. **REASON** _____
 54. **REMARKS** _____
 55. **SIGNATURE** _____
 56. **DATE** _____
 57. **TIME** _____
 58. **LOCATION** _____
 59. **REASON** _____
 60. **REMARKS** _____
 61. **SIGNATURE** _____
 62. **DATE** _____
 63. **TIME** _____
 64. **LOCATION** _____
 65. **REASON** _____
 66. **REMARKS** _____
 67. **SIGNATURE** _____
 68. **DATE** _____
 69. **TIME** _____
 70. **LOCATION** _____
 71. **REASON** _____
 72. **REMARKS** _____
 73. **SIGNATURE** _____
 74. **DATE** _____
 75. **TIME** _____
 76. **LOCATION** _____
 77. **REASON** _____
 78. **REMARKS** _____
 79. **SIGNATURE** _____
 80. **DATE** _____
 81. **TIME** _____
 82. **LOCATION** _____
 83. **REASON** _____
 84. **REMARKS** _____
 85. **SIGNATURE** _____
 86. **DATE** _____
 87. **TIME** _____
 88. **LOCATION** _____
 89. **REASON** _____
 90. **REMARKS** _____
 91. **SIGNATURE** _____
 92. **DATE** _____
 93. **TIME** _____
 94. **LOCATION** _____
 95. **REASON** _____
 96. **REMARKS** _____
 97. **SIGNATURE** _____
 98. **DATE** _____
 99. **TIME** _____
 100. **LOCATION** _____
 101. **REASON** _____
 102. **REMARKS** _____
 103. **SIGNATURE** _____
 104. **DATE** _____
 105. **TIME** _____
 106. **LOCATION** _____
 107. **REASON** _____
 108. **REMARKS** _____
 109. **SIGNATURE** _____
 110. **DATE** _____
 111. **TIME** _____
 112. **LOCATION** _____
 113. **REASON** _____
 114. **REMARKS** _____
 115. **SIGNATURE** _____
 116. **DATE** _____
 117. **TIME** _____
 118. **LOCATION** _____
 119. **REASON** _____
 120. **REMARKS** _____
 121. **SIGNATURE** _____
 122. **DATE** _____
 123. **TIME** _____
 124. **LOCATION** _____
 125. **REASON** _____
 126. **REMARKS** _____
 127. **SIGNATURE** _____
 128. **DATE** _____
 129. **TIME** _____
 130. **LOCATION** _____
 131. **REASON** _____
 132. **REMARKS** _____
 133. **SIGNATURE** _____
 134. **DATE** _____
 135. **TIME** _____
 136. **LOCATION** _____
 137. **REASON** _____
 138. **REMARKS** _____
 139. **SIGNATURE** _____
 140. **DATE** _____
 141. **TIME** _____
 142. **LOCATION** _____
 143. **REASON** _____
 144. **REMARKS** _____
 145. **SIGNATURE** _____
 146. **DATE** _____
 147. **TIME** _____
 148. **LOCATION** _____
 149. **REASON** _____
 150. **REMARKS** _____
 151. **SIGNATURE** _____
 152. **DATE** _____
 153. **TIME** _____
 154. **LOCATION** _____
 155. **REASON** _____
 156. **REMARKS** _____
 157. **SIGNATURE** _____
 158. **DATE** _____
 159. **TIME** _____
 160. **LOCATION** _____
 161. **REASON** _____
 162. **REMARKS** _____
 163. **SIGNATURE** _____
 164. **DATE** _____
 165. **TIME** _____
 166. **LOCATION** _____
 167. **REASON** _____
 168. **REMARKS** _____
 169. **SIGNATURE** _____
 170. **DATE** _____
 171. **TIME** _____
 172. **LOCATION** _____
 173. **REASON** _____
 174. **REMARKS** _____
 175. **SIGNATURE** _____
 176. **DATE** _____
 177. **TIME** _____
 178. **LOCATION** _____
 179. **REASON** _____
 180. **REMARKS** _____
 181. **SIGNATURE** _____
 182. **DATE** _____
 183. **TIME** _____
 184. **LOCATION** _____
 185. **REASON** _____
 186. **REMARKS** _____
 187. **SIGNATURE** _____
 188. **DATE** _____
 189. **TIME** _____
 190. **LOCATION** _____
 191. **REASON** _____
 192. **REMARKS** _____
 193. **SIGNATURE** _____
 194. **DATE** _____
 195. **TIME** _____
 196. **LOCATION** _____
 197. **REASON** _____
 198. **REMARKS** _____
 199. **SIGNATURE** _____
 200. **DATE** _____
 201. **TIME** _____
 202. **LOCATION** _____
 203. **REASON** _____
 204. **REMARKS** _____
 205. **SIGNATURE** _____
 206. **DATE** _____
 207. **TIME** _____
 208. **LOCATION** _____
 209. **REASON** _____
 210. **REMARKS** _____
 211. **SIGNATURE** _____
 212. **DATE** _____
 213. **TIME** _____
 214. **LOCATION** _____
 215. **REASON** _____
 216. **REMARKS** _____
 217. **SIGNATURE** _____
 218. **DATE** _____
 219. **TIME** _____
 220. **LOCATION** _____
 221. **REASON** _____
 222. **REMARKS** _____
 223. **SIGNATURE** _____
 224. **DATE** _____
 225. **TIME** _____
 2

[illegible]

Penn State Harrisburg

1. $\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$

[illegible][illegible]

18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713



Figure 1: Schematic representation of the experimental design. The diagram shows a sequence of events: a fixation cross (0.5 s), a stimulus (0.5 s), and a response (0.5 s). The stimulus is a 10-item list of words, with the first item being 'the' and the last item being 'the'. The response is a single word, 'the'.

1. 在 1990 年 12 月 1 日以前， $\text{M}(\text{C}_{10}\text{H}_8) = \text{M}(\text{C}_{10}\text{H}_8)_{\text{C}_{10}\text{H}_8}$ 的
 2. 在 1990 年 12 月 1 日以后， $\text{M}(\text{C}_{10}\text{H}_8) = \text{M}(\text{C}_{10}\text{H}_8)_{\text{C}_{10}\text{H}_8}$ 的
 3. 在 1990 年 12 月 1 日以后， $\text{M}(\text{C}_{10}\text{H}_8) = \text{M}(\text{C}_{10}\text{H}_8)_{\text{C}_{10}\text{H}_8}$ 的
 4. 在 1990 年 12 月 1 日以后， $\text{M}(\text{C}_{10}\text{H}_8) = \text{M}(\text{C}_{10}\text{H}_8)_{\text{C}_{10}\text{H}_8}$ 的
 5. 在 1990 年 12 月 1 日以后， $\text{M}(\text{C}_{10}\text{H}_8) = \text{M}(\text{C}_{10}\text{H}_8)_{\text{C}_{10}\text{H}_8}$ 的
 6. 在 1990 年 12 月 1 日以后， $\text{M}(\text{C}_{10}\text{H}_8) = \text{M}(\text{C}_{10}\text{H}_8)_{\text{C}_{10}\text{H}_8}$ 的
 7. 在 1990 年 12 月 1 日以后， $\text{M}(\text{C}_{10}\text{H}_8) = \text{M}(\text{C}_{10}\text{H}_8)_{\text{C}_{10}\text{H}_8}$ 的
 8. 在 1990 年 12 月 1 日以后， $\text{M}(\text{C}_{10}\text{H}_8) = \text{M}(\text{C}_{10}\text{H}_8)_{\text{C}_{10}\text{H}_8}$ 的
 9. 在 1990 年 12 月 1 日以后， $\text{M}(\text{C}_{10}\text{H}_8) = \text{M}(\text{C}_{10}\text{H}_8)_{\text{C}_{10}\text{H}_8}$ 的
 10. 在 1990 年 12 月 1 日以后， $\text{M}(\text{C}_{10}\text{H}_8) = \text{M}(\text{C}_{10}\text{H}_8)_{\text{C}_{10}\text{H}_8}$ 的

[illegible][illegible]

| | |
|--|-----------|
| Part of the following table is the same as the one in the previous page. | |
| Population | |
| 1950 | 1,200,000 |
| 1951 | 1,250,000 |
| 1952 | 1,300,000 |
| 1953 | 1,350,000 |
| 1954 | 1,400,000 |
| 1955 | 1,450,000 |
| 1956 | 1,500,000 |
| 1957 | 1,550,000 |
| 1958 | 1,600,000 |
| 1959 | 1,650,000 |
| 1960 | 1,700,000 |
| 1961 | 1,750,000 |
| 1962 | 1,800,000 |
| 1963 | 1,850,000 |
| 1964 | 1,900,000 |
| 1965 | 1,950,000 |
| 1966 | 2,000,000 |
| 1967 | 2,050,000 |
| 1968 | 2,100,000 |
| 1969 | 2,150,000 |
| 1970 | 2,200,000 |
| 1971 | 2,250,000 |
| 1972 | 2,300,000 |
| 1973 | 2,350,000 |
| 1974 | 2,400,000 |
| 1975 | 2,450,000 |
| 1976 | 2,500,000 |
| 1977 | 2,550,000 |
| 1978 | 2,600,000 |
| 1979 | 2,650,000 |
| 1980 | 2,700,000 |
| 1981 | 2,750,000 |
| 1982 | 2,800,000 |
| 1983 | 2,850,000 |
| 1984 | 2,900,000 |
| 1985 | 2,950,000 |
| 1986 | 3,000,000 |
| 1987 | 3,050,000 |
| 1988 | 3,100,000 |
| 1989 | 3,150,000 |
| 1990 | 3,200,000 |
| 1991 | 3,250,000 |
| 1992 | 3,300,000 |
| 1993 | 3,350,000 |
| 1994 | 3,400,000 |
| 1995 | 3,450,000 |
| 1996 | 3,500,000 |
| 1997 | 3,550,000 |
| 1998 | 3,600,000 |
| 1999 | 3,650,000 |
| 2000 | 3,700,000 |
| 2001 | 3,750,000 |
| 2002 | 3,800,000 |
| 2003 | 3,850,000 |
| 2004 | 3,900,000 |
| 2005 | 3,950,000 |
| 2006 | 4,000,000 |
| 2007 | 4,050,000 |
| 2008 | 4,100,000 |
| 2009 | 4,150,000 |
| 2010 | 4,200,000 |
| 2011 | 4,250,000 |
| 2012 | 4,300,000 |
| 2013 | 4,350,000 |
| 2014 | 4,400,000 |
| 2015 | 4,450,000 |
| 2016 | 4,500,000 |
| 2017 | 4,550,000 |
| 2018 | 4,600,000 |
| 2019 | 4,650,000 |
| 2020 | 4,700,000 |
| 2021 | 4,750,000 |
| 2022 | 4,800,000 |
| 2023 | 4,850,000 |
| 2024 | 4,900,000 |
| 2025 | 4,950,000 |
| 2026 | 5,000,000 |
| 2027 | 5,050,000 |
| 2028 | 5,100,000 |
| 2029 | 5,150,000 |
| 2030 | 5,200,000 |
| 2031 | 5,250,000 |
| 2032 | 5,300,000 |
| 2033 | 5,350,000 |
| 2034 | 5,400,000 |
| 2035 | 5,450,000 |
| 2036 | 5,500,000 |
| 2037 | 5,550,000 |
| 2038 | 5,600,000 |
| 2039 | 5,650,000 |
| 2040 | 5,700,000 |
| 2041 | 5,750,000 |
| 2042 | 5,800,000 |
| 2043 | 5,850,000 |
| 2044 | 5,900,000 |
| 2045 | 5,950,000 |
| 2046 | 6,000,000 |
| 2047 | 6,050,000 |
| 2048 | 6,100,000 |
| 2049 | 6,150,000 |
| 2050 | 6,200,000 |
| 2051 | 6,250,000 |
| 2052 | 6,300,000 |
| 2053 | 6,350,000 |
| 2054 | 6,400,000 |
| 2055 | 6,450,000 |
| 2056 | 6,500,000 |
| 2057 | 6,550,000 |
| 2058 | 6,600,000 |
| 2059 | 6,650,000 |
| 2060 | 6,700,000 |
| 2061 | 6,750,000 |
| 2062 | 6,800,000 |
| 2063 | 6,850,000 |
| 2064 | 6,900,000 |
| 2065 | 6,950,000 |
| 2066 | 7,000,000 |
| 2067 | 7,050,000 |
| 2068 | 7,100,000 |
| 2069 | 7,150,000 |
| 2070 | 7,200,000 |
| 2071 | 7,250,000 |
| 2072 | 7,300,000 |
| 2073 | 7,350,000 |
| 2074 | 7,400,000 |
| 2075 | 7,450,000 |
| 2076 | 7,500,000 |
| 2077 | 7,550,000 |
| 2078 | 7,600,000 |
| 2079 | 7,650,000 |
| 2080 | 7,700,000 |
| 2081 | 7,750,000 |
| 2082 | 7,800,000 |
| 2083 | 7,850,000 |
| 2084 | 7,900,000 |
| 2085 | 7,950,000 |
| 2086 | 8,000,000 |
| 2087 | 8,050,000 |
| 2088 | 8,100,000 |
| 2089 | 8,150,000 |
| 2090 | 8,200,000 |
| 2091 | 8,250,000 |
| 2092 | 8,300,000 |
| 2093 | 8,350,000 |
| 2094 | 8,400,000 |
| 2095 | 8,450,000 |
| 2096 | 8,500,000 |
| 2097 | 8,550,000 |
| 2098 | 8,600,000 |
| 2099 | 8,650,000 |
| 2100 | 8,700,000 |

NAME: 00427, L. n. caesi, 3. 1910, 1911, 1912, 1913, 1914, 1915, 1916, 1917, 1918, 1919, 1920, 1921, 1922, 1923, 1924, 1925, 1926, 1927, 1928, 1929, 1930, 1931, 1932, 1933, 1934, 1935, 1936, 1937, 1938, 1939, 1940, 1941, 1942, 1943, 1944, 1945, 1946, 1947, 1948, 1949, 1950, 1951, 1952, 1953, 1954, 1955, 1956, 1957, 1958, 1959, 1960, 1961, 1962, 1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024, 2025, 2026, 2027, 2028, 2029, 2030, 2031, 2032, 2033, 2034, 2035, 2036, 2037, 2038, 2039, 2040, 2041, 2042, 2043, 2044, 2045, 2046, 2047, 2048, 2049, 2050, 2051, 2052, 2053, 2054, 2055, 2056, 2057, 2058, 2059, 2060, 2061, 2062, 2063, 2064, 2065, 2066, 2067, 2068, 2069, 2070, 2071, 2072, 2073, 2074, 2075, 2076, 2077, 2078, 2079, 2080, 2081, 2082, 2083, 2084, 2085, 2086, 2087, 2088, 2089, 2090, 2091, 2092, 2093, 2094, 2095, 2096, 2097, 2098, 2099, 2100, 2101, 2102, 2103, 2104, 2105, 2106, 2107, 2108, 2109, 2110, 2111, 2112, 2113, 2114, 2115, 2116, 2117, 2118, 2119, 2120, 2121, 2122, 2123, 2124, 2125, 2126, 2127, 2128, 2129, 2130, 2131, 2132, 2133, 2134, 2135, 2136, 2137, 2138, 2139, 2140, 2141, 2142, 2143, 2144, 2145, 2146, 2147, 2148, 2149, 2150, 2151, 2152, 2153, 2154, 2155, 2156, 2157, 2158, 2159, 2160, 2161, 2162, 2163, 2164, 2165, 2166, 2167, 2168, 2169, 2170, 2171, 2172, 2173, 2174, 2175, 2176, 2177, 2178, 2179, 2180, 2181, 2182, 2183, 2184, 2185, 2186, 2187, 2188, 2189, 2190, 2191, 2192, 2193, 2194, 2195, 2196, 2197, 2198, 2199, 2200, 2201, 2202, 2203, 2204, 2205, 2206, 2207, 2208, 2209, 2210, 2211, 2212, 2213, 2214, 2215, 2216, 2217, 2218, 2219, 2220, 2221, 2222, 2223, 2224, 2225, 2226, 2227, 2228, 2229, 2230, 2231, 2232, 2233, 2234, 2235, 2236, 2237, 2238, 2239, 2240, 2241, 2242, 2243, 2244, 2245, 2246, 2247, 2248, 2249, 2250, 2251, 2252, 2253, 2254, 2255, 2256, 2257, 2258, 2259, 2260, 2261, 2262, 2263, 2264, 2265, 2266, 2267, 2268, 2269, 2270, 2271, 2272, 2273, 2274, 2275, 2276, 2277, 2278, 2279, 2280, 2281, 2282, 2283, 2284, 2285, 2286, 2287, 2288, 2289, 2290, 2291, 2292, 2293, 2294, 2295, 2296, 2297, 2298, 2299, 2300, 2301, 2302, 2303, 2304, 2305, 2306, 2307, 2308, 2309, 2310, 2311, 2312, 2313, 2314, 2315, 2316, 2317, 2318, 2319, 2320, 2321, 2322, 2323, 2324, 2325, 2326, 2327, 2328, 2329, 2330, 2331, 2332, 2333, 2334, 2335, 2336, 2337, 2338, 2339, 2340, 2341, 2342, 2343, 2344, 2345, 2346, 2347, 2348, 2349, 2350, 2351, 2352, 2353, 2354, 2355, 2356, 2357, 2358, 2359, 2360, 2361, 2362, 2363, 2364, 2365, 2366, 2367, 2368, 2369, 2370, 2371, 2372, 2373, 2374, 2375, 2376, 2377, 2378, 2379, 2380, 2381, 2382, 2383, 2384, 2385, 2386, 2387, 2388, 2389, 2390, 2391, 2392, 2393, 2394, 2395, 2396, 2397, 2398, 2399, 2400, 2401, 2402, 2403, 2404, 2405, 2406, 2407, 2408, 2409, 2410, 2411, 2412, 2413, 2414, 2415, 2416, 2417, 2418, 2419, 2420, 2421, 2422, 2423, 2424, 2425, 2426, 2427, 2428, 2429, 2430, 2431, 2432, 2433, 2434, 2435, 2436, 2437, 2438, 2439, 2440, 2441, 2442, 2443, 2444, 2445, 2446, 2447, 2448, 2449, 2450, 2451, 2452, 2453, 2454, 2455, 2456, 2457, 2458, 2459, 2460, 2461, 2462, 2463, 2464, 2465, 2466, 2467, 2468, 2469, 2470, 2471, 2472, 2473, 2474, 2475, 2476, 2477, 2478, 2479, 2480, 2481, 2482, 2483, 2484, 2485, 2486, 2487, 2488, 2489, 2490, 2491, 2492, 2493, 2494, 2495, 2496, 2497, 2498, 2499, 2500, 2501, 2502, 2503, 2504, 2505, 2506, 2507, 2508, 2509, 2510, 2511, 2512, 2513, 2514, 2515, 2516, 2517, 2518, 2519, 2520, 2521, 2522, 2523, 2524, 2525, 2526, 2527, 2528, 2529, 2530, 2531, 2532, 2533, 2534, 2535, 2536, 2537, 2538, 2539, 2540, 2541, 2542, 2543, 2544, 2545, 2546, 2547, 2548, 2549, 2550, 2551, 2552, 2553, 2554, 2555, 2556, 2557, 2558, 2559, 2560, 2561, 2562, 2563, 2564, 2565, 2566, 2567, 2568, 2569, 2570, 2571, 2572, 2573, 2574, 2575, 2576, 2577, 2578, 2579, 2580, 2581, 2582, 2583, 2584, 2585, 2586, 2587, 2588,

[illegible]

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|------------|-----|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| Group Name | Pin | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 | 128 | 129 | 130 | 131 | 132 | 133 | 134 | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 142 | 143 | 144 | 145 | 146 | 147 | 148 | 149 | 150 | 151 | 152 | 153 | 154 | 155 | 156 | 157 | 158 | 159 | 160 | 161 | 162 | 163 | 164 | 165 | 166 | 167 | 168 | 169 | 170 | 171 | 172 | 173 | 174 | 175 | 176 | 177 | 178 | 179 | 180 | 181 | 182 | 183 | 184 | 185 | 186 | 187 | 188 | 189 | 190 | 191 | 192 | 193 | 194 | 195 | 196 | 197 | 198 | 199 | 200 | 201 | 202 | 203 | 204 | 205 | 206 | 207 | 208 | 209 | 210 | 211 | 212 | 213 | 214 | 215 | 216 | 217 | 218 | 219 | 220 | 221 | 222 | 223 | 224 | 225 | 226 | 227 | 228 | 229 | 230 | 231 | 232 | 233 | 234 | 235 | 236 | 237 | 238 | 239 | 240 | 241 | 242 | 243 | 244 | 245 | 246 | 247 | 248 | 249 | 250 | 251 | 252 | 253 | 254 | 255 | 256 | 257 | 258 | 259 | 260 | 261 | 262 | 263 | 264 | 265 | 266 | 267 | 268 | 269 | 270 | 271 | 272 | 273 | 274 | 275 | 276 | 277 | 278 | 279 | 280 | 281 | 282 | 283 | 284 | 285 | 286 | 287 | 288 | 289 | 290 | 291 | 292 | 293 | 294 | 295 | 296 | 297 | 298 | 299 | 300 | 301 | 302 | 303 | 304 | 305 | 306 | 307 | 308 | 309 | 310 | 311 | 312 | 313 | 314 | 315 | 316 | 317 | 318 | 319 | 320 | 321 | 322 | 323 | 324 | 325 | 326 | 327 | 328 | 329 | 330 | 331 | 332 | 333 | 334 | 335 | 336 | 337 | 338 | 339 | 340 | 341 | 342 | 343 | 344 | 345 | 346 | 347 | 348 | 349 | 350 | 351 | 352 | 353 | 354 | 355 | 356 | 357 | 358 | 359 | 360 | 361 | 362 | 363 | 364 | 365 | 366 | 367 | 368 | 369 | 370 | 371 | 372 | 373 | 374 | 375 | 376 | 377 | 378 | 379 | 380 | 381 | 382 | 383 | 384 | 385 | 386 | 387 | 388 | 389 | 390 | 391 | 392 | 393 | 394 | 395 | 396 | 397 | 398 | 399 | 400 | 401 | 402 | 403 | 404 | 405 | 406 | 407 | 408 | 409 | 410 | 411 | 412 | 413 | 414 | 415 | 416 | 417 | 418 | 419 | 420 | 421 | 422 | 423 | 424 | 425 | 426 | 427 | 428 | 429 | 430 | 431 | 432 | 433 | 434 | 435 | 436 | 437 | 438 | 439 | 440 | 441 | 442 | 443 | 444 | 445 | 446 | 447 | 448 | 449 | 450 | 451 | 452 | 453 | 454 | 455 | 456 | 457 | 458 | 459 | 460 | 461 | 462 | 463 | 464 | 465 | 466 | 467 | 468 | 469 | 470 | 471 | 472 | 473 | 474 | 475 | 476 | 477 | 478 | 479 | 480 | 481 | 482 | 483 | 484 | 485 | 486 | 487 | 488 | 489 | 490 | 491 | 492 | 493 | 494 | 495 | 496 | 497 | 498 | 499 | 500 | 501 | 502 | 503 | 504 | 505 | 506 | 507 | 508 | 509 | 510 | 511 | 512 | 513 | 514 | 515 | 516 | 517 | 518 | 519 | 520 | 521 | 522 | 523 | 524 | 525 | 526 | 527 | 528 | 529 | 530 | 531 | 532 | 533 | 534 | 535 | 536 | 537 | 538 | 539 | 540 | 541 | 542 | 543 | 544 | 545 | 546 | 547 | 548 | 549 | 550 | 551 | 552 | 553 | 554 | 555 | 556 | 557 | 558 | 559 | 560 | 561 | 562 | 563 | 564 | 565 | 566 | 567 | 568 | 569 | 570 | 571 | 572 | 573 | 574 | 575 | 576 | 577 | 578 | 579 | 580 | 581 | 582 | 583 | 584 | 585 | 586 | 587 | 588 | 589 | 590 | 591 | 592 | 593 | 594 | 595 | 596 | 597 | 598 | 599 | 600 | 601 | 602 | 603 | 604 | 605 | 606 | 607 | 608 | 609 | 610 | 611 | 612 | 613 | 614 | 615 | 616 | 617 | 618 | 619 | 620 | 621 | 622 | 623 | 624 | 625 | 626 | 627 | 628 | 629 | 630 | 631 | 632 | 633 | 634 | 635 | 636 | 637 | 638 | 639 | 640 | 641 | 642 | 643 | 644 | 645 | 646 | 647 | 648 | 649 | 650 | 651 | 652 | 653 | 654 | 655 | 656 | 657 | 658 | 659 | 660 | 661 | 662 | 663 | 664 | 665 | 666 | 667 | 668 | 669 | 670 | 671 | 672 | 673 | 674 | 675 | 676 | 677 | 678 | 679 | 680 | 681 | 682 | 683 | 684 | 685 | 686 | 687 | 688 | 689 | 690 | 691 | 692 | 693 | 694 | 695 | 696 | 697 | 698 | 699 | 700 | 701 | 702 | 703 | 704 | 705 | 706 | 707 | 708 | 709 | 710 | 711 | 712 | 713 | 714 | 715 | 716 | 717 | 718 | 719 | 720 | 721 | 722 | 723 | 724 | 725 | 726 | 727 | 728 | 729 | 730 | 731 | 732 | 733 | 734 | 735 | 736 | 737 | 738 | 739 | 740 | 741 | 742 | 743 | 744 | 745 | 746 | 747 | 748 | 749 | 750 | 751 | 752 | 753 | 754 | 755 | 756 | 757 | 758 | 759 | 760 | 761 | 762 | 763 | 764 | 765 | 766 | 767 | 768 | 769 | 770 | 771 | 772 | 773 | 774 | 775 | 776 | 777 | 778 | 779 | 780 | 781 | 782 | 783 | 784 | 785 | 786 | 787 | 788 | 789 | 790 | 791 | 792 | 793 | 794 | 795 | 796 | 797 | 798 | 799 | 800 | 801 | 802 | 803 | 804 | 805 | 806 | 807 | 808 | 809 | 810 | 811 | 812 | 813 | 814 | 815 | 816 | 817 | 818 | 819 | 820 | 821 | 822 | 823 | 824 | 825 | 826 | 827 | 828 | 829 | 830 | 831 | 832 | 833 | 834 | 835 | 836 | 837 | 838 | 839 | 840 | 841 | 842 | 843 | 844 | 845 | 846 | 847 | 848 | 849 | 850 | 851 | 852 | 853 | 854 | 855 | 856 | 857 | 858 | 859 | 860 | 861 | 862 | 863 | 864 | 865 | 866 | 867 | 868 | 869 | 870 | 871 | 872 | 873 | 874 | 875 | 876 | 877 | 878 | 879 | 880 | 881 | 882 | 883 | 884 | 885 | 886 | 887 | 888 | 889 | 890 | 891 | 892 | 893 | 894 | 895 | 896 | 897 | 898 | 899 | 900 | 901 | 902 | 903 | 904 | 905 | 906 | 907 | 908 | 909 | 910 | 911 | 912 | 913 | 914 | 915 | 916 | 917 | 918 | 919 | 920 | 921 | 922 | 923 | 924 | 925 | 926 | 927 | 928 | 929 | 930 | 931 | 932 | 933 | 934 | 935 | 936 | 937 | 938 | 939 | 940 | 941 | 942 | 943 | 944 | 945 | 946 | 947 | 948 | 949 | 950 | 951 | 952 | 953 | 954 | 955 | 956 | 957 | 958 | 959 | 960 | 961 | 962 | 963 | 964 | 965 | 966 | 967 | 968 | 969 | 970 | 971 | 972 | 973 | 974 | 975 | 976 | 977 | 978 | 979 | 980 | 981 | 982 | 983 | 984 | 985 | 986 | 987 | 988 | 989 | 990 | 991 | 992 | 993 | 994 | 995 | 996 | 997 | 998 | 999 | 1000 | 1001 | 1002 | 1003 | 1004 | 1005 | 1006 | 1007 | 1008 | 1009 | 1010 | 1011 | 1012 | 1013 | 1014 | 1015 | 1016 | 1017 | 1018 | 1019 | 1020 | 1021 | 1022 | 1023 | 1024 | 1025 | 1026 | 1027 | 1028 | 1029 | 1030 | 1031 | 1032 | 1033 | 1034 | 1035 | 1036 | 1037 | 1038 | 1039 | 1040 | 1041 | 1042 | 1043 | 1044 | 1045 | 1046 | 1047 | 1048 | 1049 | 1050 | 1051 | 1052 | 1053 | 1054 | 1055 | 1056 | 1057 | 1058 | 1059 | 1060 | 1061 | 1062 | 1063 | 1064 | 1065 | 1066 | 1067 | 1068 | 1069 | 1070 | 1071 | 1072 | 1073 | 1074 | 1075 | 1076 | 1077 | 1078 | 1079 | 1080 | 1081 | 1082 | 1083 | 1084 | 1085 | 1086 | 1087 | 1088 | 1089 | 1090 | 1091 | 1092 | 1093 | 1094 | 1095 | 1096 | 1097 | 1098 | 1099 | 1100 | 1101 | 1102 | 1103 | 1104 | 1105 | 1106 | 1107 | 1108 | 1109 | 1110 | 1111 | 1112 | 1113 | 1114 | 1115 | 1116 | 1117 | 1118 | 1119 | 1120 | 1121 | 1122 | 1123 | 1124 | 1125 | 1126 | 1127 | 1128 | 1129 | 1130 | 1131 | 1132 | 1133 | 1134 | 1135 | 1136 | 1137 | 1138 | 1139 | 1140 | 1141 | 1142 | 1143 | 1144 | 1145 | 1146 | 1147 | 1148 | 1149 | 1150 | 1151 | 1152 | 1153 | 1154 | 1155 | 1156 | 1157 | 1158 | 1159 | 1160 | 1161 | 1162 | 1163 | 1164 | 1165 | 1166 | 1167 | 1168 | 1169 | 1170 | 1171 | 1172 | 1173 | 1174 | 1175 | 1176 | 1177 | 1178 | 1179 | 1180 | 1181 | 1182 | 1183 | 1184 | 1185 | 1186 | 1187 | 1188 | 1189 | 1190 | 1191 | 1192 | 1193 | 1194 | 1195 | 1196 | 1197 | 1198 | 1199 | 1200 | 1201 | 1202 | 1203 | 1204 | 1205 | 1206 | 1207 | 1208 | 1209 | 1210 | 1211 | 1212 | 1213 | 1214 | 1215 | 1216 | 1217 | 1218 | 1219 | 1220 | 1221 | 1222 | 1223 | 1224 | 1225 | 1226 | 1227 | 1228 | 1229 | 1230 | 1231 | 1232 | 1233 | 1234 | 1235 | 1236 | 1237 | 1238 | 1239 | 1240 | 1241 | 1242 | 1243 | 1244 | 1245 | 1246 | 1247 | 1248 | 1249 | 1250 | 1251 | 1252 | 1253 | 1254 | 1255 | 1256 | 1257 | 1258 | 1259 | 1260 | 1261 | 1262 | 1263 | 1264 | 1265 | 1266 | 1267 | 1268 | 1269 | 1270 | 1271 | 1272 | 1273 | 1274 | 1275 | 1276 | 1277 | 1278 | 1279 | 1280 | 1281 | 1282 | 1283 | 1284 | 1285 | 1286 | 1287 | 1288 | 1289 | 1290 | 1291 | 1292 | 1293 | 1294 | 1295 | 1296 | 1297 | 1298 | 1299 | 1300 | 1301 | 1302 | 1303 | 1304 | 1305 | 1306 | 1307 | 1308 | 1309 | 1310 | 1311 | 1312 | 1313 | 1314 | 1315 | 1316 | 1317 | 1318 | 1319 | 1320 | 1321 | 1322 | 1323 | 1324 | 1325 | 1326 | 1327 | 1328 | 1329 | 1330 | 1331 | 1332 | 1333 | 1334 | 1335 | 1336 | 1337 | 1338 | 1339 | 1340 | 1341 | 1342 | 1343 | 1344 | 1345 | 1346 | 1347 | 1348 | 1349 | 1350 | 1351 | 1352 | 1353 | 1354 | 1355 | 1356 | 1357 | 1358 | 1359 | 1360 | 1361 | 1362 | 1363 | 1364 | 1365 | 1366 | 1367 | 1368 | 1369 | 1370 | 1371 | 1372 | 1373 | 1374 | 1375 | 1376 | 1377 | 1378 | 1379 | 1380 | 1381 | 1382 | 1383 | 1384 | 1385 | 1386 | 1387 | 1388 | 1389 | 1390 | 1391 | 1392 | 1393 | 1394 | 1395 | 1396 | 1397 | 1398 | 1399 | 1400 | 1401 | 1402 | 1403 | 1404 | 1405 | 1406 | 1407 | 1408 | 1409 | 1410 | 1411 | 1412 | 1413 | 1414 | 1415 | 1416 | 1417 | 1418 | 1419 | 1420 | 1421 | 1422 | 1423 | 1424 | 1425 | 1426 | 1427 | 1428 | 1429 | 1430 | 1431 | 1432 | 1433 | 1434 | 1435 | 1436 | 1437 | 1438 | 1439 | 1440 | 1441 | 1442 | 1443 | 1444 | 1445 | 1446 | 1447 | 1448 | 1449 | 1450 | 1451 | 1452 | 1453 | 1454 | 1455 | 1456 | 1457 | 1458 | 1459 | 1460 | 1461 | 1462 | 1463 | 1464 | 1465 | 1466 | 1467 | 1468 | 1469 | 1470 | 1471 | 1472 | 1473 | 1474 | 1475 | 1476 | 1477 | 1478 | 1479 | 1480 | 1481 | 1482 | 1483 | 1484 | 1485 | 1486 | 1487 | 1488 | 1489 | 1490 | 1491 | 1492 | 1493 | 1494 |
|------------|-----|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|

IN, OUT and the Next Registers

| Headline 1: Title, Date, Location, etc. | | | | | | | | | |
|---|-----|-----|-----|-----|-----|-----|-----|-----|------|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 |
| 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 |
| 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 |
| 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 |
| 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |
| 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 |
| 121 | 122 | 123 | 124 | 125 | 126 | 127 | 128 | 129 | 130 |
| 131 | 132 | 133 | 134 | 135 | 136 | 137 | 138 | 139 | 140 |
| 141 | 142 | 143 | 144 | 145 | 146 | 147 | 148 | 149 | 150 |
| 151 | 152 | 153 | 154 | 155 | 156 | 157 | 158 | 159 | 160 |
| 161 | 162 | 163 | 164 | 165 | 166 | 167 | 168 | 169 | 170 |
| 171 | 172 | 173 | 174 | 175 | 176 | 177 | 178 | 179 | 180 |
| 181 | 182 | 183 | 184 | 185 | 186 | 187 | 188 | 189 | 190 |
| 191 | 192 | 193 | 194 | 195 | 196 | 197 | 198 | 199 | 200 |
| 201 | 202 | 203 | 204 | 205 | 206 | 207 | 208 | 209 | 210 |
| 211 | 212 | 213 | 214 | 215 | 216 | 217 | 218 | 219 | 220 |
| 221 | 222 | 223 | 224 | 225 | 226 | 227 | 228 | 229 | 230 |
| 231 | 232 | 233 | 234 | 235 | 236 | 237 | 238 | 239 | 240 |
| 241 | 242 | 243 | 244 | 245 | 246 | 247 | 248 | 249 | 250 |
| 251 | 252 | 253 | 254 | 255 | 256 | 257 | 258 | 259 | 260 |
| 261 | 262 | 263 | 264 | 265 | 266 | 267 | 268 | 269 | 270 |
| 271 | 272 | 273 | 274 | 275 | 276 | 277 | 278 | 279 | 280 |
| 281 | 282 | 283 | 284 | 285 | 286 | 287 | 288 | 289 | 290 |
| 291 | 292 | 293 | 294 | 295 | 296 | 297 | 298 | 299 | 300 |
| 301 | 302 | 303 | 304 | 305 | 306 | 307 | 308 | 309 | 310 |
| 311 | 312 | 313 | 314 | 315 | 316 | 317 | 318 | 319 | 320 |
| 321 | 322 | 323 | 324 | 325 | 326 | 327 | 328 | 329 | 330 |
| 331 | 332 | 333 | 334 | 335 | 336 | 337 | 338 | 339 | 340 |
| 341 | 342 | 343 | 344 | 345 | 346 | 347 | 348 | 349 | 350 |
| 351 | 352 | 353 | 354 | 355 | 356 | 357 | 358 | 359 | 360 |
| 361 | 362 | 363 | 364 | 365 | 366 | 367 | 368 | 369 | 370 |
| 371 | 372 | 373 | 374 | 375 | 376 | 377 | 378 | 379 | 380 |
| 381 | 382 | 383 | 384 | 385 | 386 | 387 | 388 | 389 | 390 |
| 391 | 392 | 393 | 394 | 395 | 396 | 397 | 398 | 399 | 400 |
| 401 | 402 | 403 | 404 | 405 | 406 | 407 | 408 | 409 | 410 |
| 411 | 412 | 413 | 414 | 415 | 416 | 417 | 418 | 419 | 420 |
| 421 | 422 | 423 | 424 | 425 | 426 | 427 | 428 | 429 | 430 |
| 431 | 432 | 433 | 434 | 435 | 436 | 437 | 438 | 439 | 440 |
| 441 | 442 | 443 | 444 | 445 | 446 | 447 | 448 | 449 | 450 |
| 451 | 452 | 453 | 454 | 455 | 456 | 457 | 458 | 459 | 460 |
| 461 | 462 | 463 | 464 | 465 | 466 | 467 | 468 | 469 | 470 |
| 471 | 472 | 473 | 474 | 475 | 476 | 477 | 478 | 479 | 480 |
| 481 | 482 | 483 | 484 | 485 | 486 | 487 | 488 | 489 | 490 |
| 491 | 492 | 493 | 494 | 495 | 496 | 497 | 498 | 499 | 500 |
| 501 | 502 | 503 | 504 | 505 | 506 | 507 | 508 | 509 | 510 |
| 511 | 512 | 513 | 514 | 515 | 516 | 517 | 518 | 519 | 520 |
| 521 | 522 | 523 | 524 | 525 | 526 | 527 | 528 | 529 | 530 |
| 531 | 532 | 533 | 534 | 535 | 536 | 537 | 538 | 539 | 540 |
| 541 | 542 | 543 | 544 | 545 | 546 | 547 | 548 | 549 | 550 |
| 551 | 552 | 553 | 554 | 555 | 556 | 557 | 558 | 559 | 560 |
| 561 | 562 | 563 | 564 | 565 | 566 | 567 | 568 | 569 | 570 |
| 571 | 572 | 573 | 574 | 575 | 576 | 577 | 578 | 579 | 580 |
| 581 | 582 | 583 | 584 | 585 | 586 | 587 | 588 | 589 | 590 |
| 591 | 592 | 593 | 594 | 595 | 596 | 597 | 598 | 599 | 600 |
| 601 | 602 | 603 | 604 | 605 | 606 | 607 | 608 | 609 | 610 |
| 611 | 612 | 613 | 614 | 615 | 616 | 617 | 618 | 619 | 620 |
| 621 | 622 | 623 | 624 | 625 | 626 | 627 | 628 | 629 | 630 |
| 631 | 632 | 633 | 634 | 635 | 636 | 637 | 638 | 639 | 640 |
| 641 | 642 | 643 | 644 | 645 | 646 | 647 | 648 | 649 | 650 |
| 651 | 652 | 653 | 654 | 655 | 656 | 657 | 658 | 659 | 660 |
| 661 | 662 | 663 | 664 | 665 | 666 | 667 | 668 | 669 | 670 |
| 671 | 672 | 673 | 674 | 675 | 676 | 677 | 678 | 679 | 680 |
| 681 | 682 | 683 | 684 | 685 | 686 | 687 | 688 | 689 | 690 |
| 691 | 692 | 693 | 694 | 695 | 696 | 697 | 698 | 699 | 700 |
| 701 | 702 | 703 | 704 | 705 | 706 | 707 | 708 | 709 | 710 |
| 711 | 712 | 713 | 714 | 715 | 716 | 717 | 718 | 719 | 720 |
| 721 | 722 | 723 | 724 | 725 | 726 | 727 | 728 | 729 | 730 |
| 731 | 732 | 733 | 734 | 735 | 736 | 737 | 738 | 739 | 740 |
| 741 | 742 | 743 | 744 | 745 | 746 | 747 | 748 | 749 | 750 |
| 751 | 752 | 753 | 754 | 755 | 756 | 757 | 758 | 759 | 760 |
| 761 | 762 | 763 | 764 | 765 | 766 | 767 | 768 | 769 | 770 |
| 771 | 772 | 773 | 774 | 775 | 776 | 777 | 778 | 779 | 780 |
| 781 | 782 | 783 | 784 | 785 | 786 | 787 | 788 | 789 | 790 |
| 791 | 792 | 793 | 794 | 795 | 796 | 797 | 798 | 799 | 800 |
| 801 | 802 | 803 | 804 | 805 | 806 | 807 | 808 | 809 | 810 |
| 811 | 812 | 813 | 814 | 815 | 816 | 817 | 818 | 819 | 820 |
| 821 | 822 | 823 | 824 | 825 | 826 | 827 | 828 | 829 | 830 |
| 831 | 832 | 833 | 834 | 835 | 836 | 837 | 838 | 839 | 840 |
| 841 | 842 | 843 | 844 | 845 | 846 | 847 | 848 | 849 | 850 |
| 851 | 852 | 853 | 854 | 855 | 856 | 857 | 858 | 859 | 860 |
| 861 | 862 | 863 | 864 | 865 | 866 | 867 | 868 | 869 | 870 |
| 871 | 872 | 873 | 874 | 875 | 876 | 877 | 878 | 879 | 880 |
| 881 | 882 | 883 | 884 | 885 | 886 | 887 | 888 | 889 | 890 |
| 891 | 892 | 893 | 894 | 895 | 896 | 897 | 898 | 899 | 900 |
| 901 | 902 | 903 | 904 | 905 | 906 | 907 | 908 | 909 | 910 |
| 911 | 912 | 913 | 914 | 915 | 916 | 917 | 918 | 919 | 920 |
| 921 | 922 | 923 | 924 | 925 | 926 | 927 | 928 | 929 | 930 |
| 931 | 932 | 933 | 934 | 935 | 936 | 937 | 938 | 939 | 940 |
| 941 | 942 | 943 | 944 | 945 | 946 | 947 | 948 | 949 | 950 |
| 951 | 952 | 953 | 954 | 955 | 956 | 957 | 958 | 959 | 960 |
| 961 | 962 | 963 | 964 | 965 | 966 | 967 | 968 | 969 | 970 |
| 971 | 972 | 973 | 974 | 975 | 976 | 977 | 978 | 979 | 980 |
| 981 | 982 | 983 | 984 | 985 | 986 | 987 | 988 | 989 | 990 |
| 991 | 992 | 993 | 994 | 995 | 996 | 997 | 998 | 999 | 1000 |

| Headline 2: Title, Date, Location, etc. | | | | | | | | | |
|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 |
| 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 |
| 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 |
| 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 |
| 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |
| 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 |
| 121 | 122 | 123 | 124 | 125 | 126 | 127 | 128 | 129 | 130 |
| 131 | 132 | 133 | 134 | 135 | 136 | 137 | 138 | 139 | 140 |
| 141 | 142 | 143 | 144 | 145 | 146 | 147 | 148 | 149 | 150 |
| 151 | 152 | 153 | 154 | 155 | 156 | 157 | 158 | 159 | 160 |
| 161 | 162 | 163 | 164 | 165 | 166 | 167 | 168 | 169 | 170 |
| 171 | 172 | 173 | 174 | 175 | 176 | 177 | 178 | 179 | 180 |
| 181 | 182 | 183 | 184 | 185 | 186 | 187 | 188 | 189 | 190 |
| 191 | 192 | 193 | 194 | 195 | 196 | 197 | 198 | 199 | 200 |
| 201 | 202 | 203 | 204 | 205 | 206 | 207 | 208 | 209 | 210 |
| 211 | 212 | 213 | 214 | 215 | 216 | 217 | 218 | 219 | 220 |
| 221 | 222 | 223 | 224 | 225 | 226 | 227 | 228 | 229 | 230 |
| 231 | 232 | 233 | 234 | 235 | 236 | 237 | 238 | 239 | 240 |
| 241 | 242 | 243 | 244 | 245 | 246 | 247 | 248 | 249 | 250 |
| 251 | 252 | 253 | 254 | 255 | 256 | 257 | 258 | 259 | 260 |
| 261 | 262 | 263 | 264 | 265 | 266 | 267 | 268 | 269 | 270 |
| 271 | 272 | 273 | 274 | 275 | 276 | 277 | 278 | 279 | 280 |
| 281 | 282 | 283 | 284 | 285 | 286 | 287 | 288 | 289 | 290 |
| 291 | 292 | 293 | 294 | 295 | 296 | 297 | 298 | 299 | 300 |
| 301 | 302 | 303 | 304 | 305 | 306 | 307 | 308 | 309 | 310 |
| 311 | 312 | 313 | 314 | 315 | 316 | 317 | 318 | 319 | 320 |
| 321 | 322 | 323 | 324 | 325 | 326 | 327 | 328 | 329 | 330 |
| 331 | 332 | 333 | 334 | 335 | 336 | 337 | 338 | 339 | 340 |
| 341 | 342 | 343 | 344 | 345 | 346 | 347 | 348 | 349 | 350 |
| 351 | 352 | 353 | 354 | 355 | 356 | 357 | 358 | 359 | 360 |
| 361 | 362 | 363 | 364 | 365 | 366 | 367 | 368 | 369 | 370 |
| 371 | 372 | 373 | 374 | 375 | 376 | 377 | 378 | 379 | 380 |
| 381 | 382 | 383 | 384 | 385 | 386 | 387 | 388 | 389 | 390 |
| 391 | 392 | 393 | 394 | 395 | 396 | 397 | 398 | 399 | 400 |
| 401 | 402 | 403 | 404 | 405 | 406 | 407 | 408 | 409 | 410 |
| 411 | 412 | 413 | 414 | 415 | 416 | 417 | 418 | 419 | 420 |
| 421 | 422 | 423 | 424 | 425 | 426 | 427 | 428 | 429 | 430 |
| 431 | 432 | 433 | 434 | 435 | 436 | 437 | 438 | 439 | 440 |
| 441 | 442 | 443 | 444 | 445 | 446 | 447 | 448 | 449 | 450 |
| 451 | 452 | 453 | 454 | 455 | 456 | 457 | 458 | 459 | 460 |
| 461 | 462 | 463 | 464 | 465 | 466 | 467 | 468 | 469 | 470 |
| 471 | 472 | 473 | 474 | 475 | 476 | 477 | 478 | 479 | 480 |
| 481 | 482 | 483 | 484 | 485 | 486 | 487 | | | |

Other port addresses

As seen in the table at the beginning of this chapter and the discussion about decoding all even addresses refer to I/O functions. You may find yourself in need to read the keyboard directly from the hardware. As mentioned part of the I/OA's function is to return the state of keypresses. The keyboard is divided in 8 half-rows of 5 keys each, each half-row having its own port address:

| Address
Decimal Hex | Bits | | | | | | | | | | Address
Decimal Hex |
|------------------------|------|----|----|----|----|----|----|----|-----|-------|------------------------|
| | D0 | D1 | D2 | D3 | D4 | D4 | D3 | D2 | D1 | D0 | |
| 61436 F7FDh | | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 61436 F7FDh |
| 64510 FBFDh | Q | W | E | R | T | Y | U | | Q | P | 57342 DFFDh |
| 66022 FDFBh | A | S | D | F | G | H | J | K | L | Enter | 49150 BFFDh |
| 65276 FEFBh | Caps | Z | X | C | V | B | N | M | Sym | Space | 32766 7FFDh |

The diagram above nicely illustrates how the keyboard matrix is separated into half-rows (separated by the thick line in the middle). Pay attention to how bits are mirrored going from the outside of the keyboard to the inside.

The address of each half-row in the diagram is calculated as $254 + 256 * 255 - 2^n$

in the form is above is the number of half-row which starts with D4, the bottom right and moves in a counterclockwise manner with each successive half-row increasing by

in the byte read in, D3 to D4 stand for each of the five keys in the given half-row. D0 for the inside key and D4 for the one nearest the middle. The bit is 0 if the key is pressed and 1 if it is not.

For example to find the value of the CAPS SHIFT key you can do

```
PRINT %IN 65276 & 01
```

Writing a value using OUT to the I/OA Port 254 FEFh, controls other hardware as well. You can drive the buzzer with D4, the MIC socket with D3, read the EAR socket with D6 and modify the BORDER colour using bits D0 D1 and D2. For example to make the border a nice magenta colour you can

```
OUT 254, %000000011
```

Port addresses 32765 7FFDh, 6109 1FFDh and 57341 DFFDh control the extra memory. Executing an OUT to these ports from NextBASIC without knowing the ramifications will nearly always cause the computer to crash losing any program and data. These ports are write-only, so you cannot determine the current state of the paging by an IN instruction. This is why the BANKM system variable is always kept up to date with the last value output on its port. Check the ias section in this chapter as well as Chapter 23 The Memory where we examine the banking system in detail.

Writing to port 65533 (FFFDh) will select a particular PSG register on the AY sound chip and writing a port 49149 (BFFDh) will send a particular value to the keybase. Reading from port 65533 (FFFDh) returns the value stored in the selected register. Judicious use of these two registers can allow sounds to be generated while NextBASIC gets on with something else.

The section that follows describes all ZX Spectrum Next specific hardware ports, addressing them is via OUT and IN commands.

5. For more info on the differences between the different versions of the NextBASIC software, see the NextBASIC User Manual.

The ZX Spectrum Next Hardware Ports List

[illegible]

[illegible]

Fig. 29-25. L401, 5P11 tube

.. .. .
5P11 tube

Fig. 29-26. L401, 5P11 tube

Fig. 29-27. L401, 5P11 tube

Fig. 29-28. L401, 5P11 tube

Fig. 29-29. L401, 5P11 tube

Fig. 29-30. L401, 5P11 tube

Fig. 29-31. L401, 5P11 tube

Fig. 29-32. L401, 5P11 tube

Fig. 29-33. L401, 5P11 tube

Fig. 29-34. L401, 5P11 tube

Fig. 29-35. L401, 5P11 tube

Fig. 29-36. L401, 5P11 tube

Fig. 29-37. L401, 5P11 tube

Fig. 29-38. L401, 5P11 tube

Fig. 29-39. L401, 5P11 tube

Fig. 29-40. L401, 5P11 tube

Fig. 29-41. L401, 5P11 tube

Fig. 29-42. L401, 5P11 tube

Fig. 29-43. L401, 5P11 tube

Fig. 29-44. L401, 5P11 tube

Fig. 29-45. L401, 5P11 tube

Fig. 29-46. L401, 5P11 tube

Fig. 29-47. L401, 5P11 tube

Fig. 29-48. L401, 5P11 tube

Fig. 29-49. L401, 5P11 tube

Fig. 29-50. L401, 5P11 tube

Fig. 29-51. L401, 5P11 tube

Fig. 29-52. L401, 5P11 tube

Fig. 29-53. L401, 5P11 tube

Fig. 29-54. L401, 5P11 tube

Fig. 29-55. L401, 5P11 tube

Fig. 29-56. L401, 5P11 tube

Fig. 29-57. L401, 5P11 tube

Fig. 29-58. L401, 5P11 tube

Fig. 29-59. L401, 5P11 tube

Fig. 29-60. L401, 5P11 tube

Fig. 29-61. L401, 5P11 tube

Fig. 29-62. L401, 5P11 tube

Fig. 29-63. L401, 5P11 tube

Fig. 29-64. L401, 5P11 tube

Fig. 29-65. L401, 5P11 tube

Fig. 29-66. L401, 5P11 tube

Fig. 29-67. L401, 5P11 tube

Fig. 29-68. L401, 5P11 tube

Fig. 29-69. L401, 5P11 tube

Fig. 29-70. L401, 5P11 tube

Fig. 29-71. L401, 5P11 tube

Fig. 29-72. L401, 5P11 tube

Fig. 29-73. L401, 5P11 tube

Fig. 29-74. L401, 5P11 tube

Fig. 29-75. L401, 5P11 tube

Fig. 29-76. L401, 5P11 tube

Fig. 29-77. L401, 5P11 tube

Fig. 29-78. L401, 5P11 tube

Fig. 29-79. L401, 5P11 tube

Fig. 29-80. L401, 5P11 tube

Fig. 29-81. L401, 5P11 tube

Fig. 29-82. L401, 5P11 tube

Fig. 29-83. L401, 5P11 tube

Fig. 29-84. L401, 5P11 tube

Fig. 29-85. L401, 5P11 tube

Fig. 29-86. L401, 5P11 tube

Fig. 29-87. L401, 5P11 tube

Fig. 29-88. L401, 5P11 tube

Fig. 29-89. L401, 5P11 tube

Fig. 29-90. L401, 5P11 tube

Fig. 29-91. L401, 5P11 tube

Fig. 29-92. L401, 5P11 tube

Fig. 29-93. L401, 5P11 tube

Fig. 29-94. L401, 5P11 tube

Fig. 29-95. L401, 5P11 tube

Fig. 29-96. L401, 5P11 tube

Fig. 29-97. L401, 5P11 tube

Fig. 29-98. L401, 5P11 tube

Fig. 29-99. L401, 5P11 tube

Fig. 29-100. L401, 5P11 tube

Port 4453, 3389, 2447, 2448, 2449, 2450, 2451, 2452, 2453, 2454, 2455, 2456, 2457, 2458, 2459, 2460, 2461, 2462, 2463, 2464, 2465, 2466, 2467, 2468, 2469, 2470, 2471, 2472, 2473, 2474, 2475, 2476, 2477, 2478, 2479, 2480, 2481, 2482, 2483, 2484, 2485, 2486, 2487, 2488, 2489, 2490, 2491, 2492, 2493, 2494, 2495, 2496, 2497, 2498, 2499, 2500, 2501, 2502, 2503, 2504, 2505, 2506, 2507, 2508, 2509, 2510, 2511, 2512, 2513, 2514, 2515, 2516, 2517, 2518, 2519, 2520, 2521, 2522, 2523, 2524, 2525, 2526, 2527, 2528, 2529, 2530, 2531, 2532, 2533, 2534, 2535, 2536, 2537, 2538, 2539, 2540, 2541, 2542, 2543, 2544, 2545, 2546, 2547, 2548, 2549, 2550, 2551, 2552, 2553, 2554, 2555, 2556, 2557, 2558, 2559, 2560, 2561, 2562, 2563, 2564, 2565, 2566, 2567, 2568, 2569, 2570, 2571, 2572, 2573, 2574, 2575, 2576, 2577, 2578, 2579, 2580, 2581, 2582, 2583, 2584, 2585, 2586, 2587, 2588, 2589, 2590, 2591, 2592, 2593, 2594, 2595, 2596, 2597, 2598, 2599, 2600, 2601, 2602, 2603, 2604, 2605, 2606, 2607, 2608, 2609, 2610, 2611, 2612, 2613, 2614, 2615, 2616, 2617, 2618, 2619, 2620, 2621, 2622, 2623, 2624, 2625, 2626, 2627, 2628, 2629, 2630, 2631, 2632, 2633, 2634, 2635, 2636, 2637, 2638, 2639, 2640, 2641, 2642, 2643, 2644, 2645, 2646, 2647, 2648, 2649, 2650, 2651, 2652, 2653, 2654, 2655, 2656, 2657, 2658, 2659, 2660, 2661, 2662, 2663, 2664, 2665, 2666, 2667, 2668, 2669, 2670, 2671, 2672, 2673, 2674, 2675, 2676, 2677, 2678, 2679, 2680, 2681, 2682, 2683, 2684, 2685, 2686, 2687, 2688, 2689, 2690, 2691, 2692, 2693, 2694, 2695, 2696, 2697, 2698, 2699, 2700, 2701, 2702, 2703, 2704, 2705, 2706, 2707, 2708, 2709, 2710, 2711, 2712, 2713, 2714, 2715, 2716, 2717, 2718, 2719, 2720, 2721, 2722, 2723, 2724, 2725, 2726, 2727, 2728, 2729, 2730, 2731, 2732, 2733, 2734, 2735, 2736, 2737, 2738, 2739, 2740, 2741, 2742, 2743, 2744, 2745, 2746, 2747, 2748, 2749, 2750, 2751, 2752, 2753, 2754, 2755, 2756, 2757, 2758, 2759, 2760, 2761, 2762, 2763, 2764, 2765, 2766, 2767, 2768, 2769, 2770, 2771, 2772, 2773, 2774, 2775, 2776, 2777, 2778, 2779, 2780, 2781, 2782, 2783, 2784, 2785, 2786, 2787, 2788, 2789, 2790, 2791, 2792, 2793, 2794, 2795, 2796, 2797, 2798, 2799, 2800, 2801, 2802, 2803, 2804, 2805, 2806, 2807, 2808, 2809, 2810, 2811, 2812, 2813, 2814, 2815, 2816, 2817, 2818, 2819, 2820, 2821, 2822, 2823, 2824, 2825, 2826, 2827, 2828, 2829, 2830, 2831, 2832, 2833, 2834, 2835, 2836, 2837, 2838, 2839, 2840, 2841, 2842, 2843, 2844, 2845, 2846, 2847, 2848, 2849, 2850, 2851, 2852, 2853, 2854, 2855, 2856, 2857, 2858, 2859, 2860, 2861, 2862, 2863, 2864, 2865, 2866, 2867, 2868, 2869, 2870, 2871, 2872, 2873, 2874, 2875, 2876, 2877, 2878, 2879, 2880, 2881, 2882, 2883, 2884, 2885, 2886, 2887, 2888, 2889, 2890, 2891, 2892, 2893, 2894, 2895, 2896, 2897, 2898, 2899, 2900, 2901, 2902, 2903, 2904, 2905, 2906, 2907, 2908, 2909, 2910, 2911, 2912, 2913, 2914, 2915, 2916, 2917, 2918, 2919, 2920, 2921, 2922, 2923, 2924, 2925, 2926, 2927, 2928, 2929, 2930, 2931, 2932, 2933, 2934, 2935, 2936, 2937, 2938, 2939, 2940, 2941, 2942, 2943, 2944, 2945, 2946, 2947, 2948, 2949, 2950, 2951, 2952, 2953, 2954, 2955, 2956, 2957, 2958, 2959, 2960, 2961, 2962, 2963, 2964, 2965, 2966, 2967, 2968, 2969, 2970, 2971, 2972, 2973, 2974, 2975, 2976, 2977, 2978, 2979, 2980, 2981, 2982, 2983, 2984, 2985, 2986, 2987, 2988, 2989, 2990, 2991, 2992, 2993, 2994, 2995, 2996, 2997, 2998, 2999, 3000, 3001, 3002, 3003, 3004, 3005, 3006, 3007, 3008, 3009, 3010, 3011, 3012, 3013, 3014, 3015, 3016, 3017, 3018, 3019, 3020, 3021, 3022, 3023, 3024, 3025, 3026, 3027, 3028, 3029, 3030, 3031, 3032, 3033, 3034, 3035, 3036, 3037, 3038, 3039, 3040, 3041, 3042, 3043, 3044, 3045, 3046, 3047, 3048, 3049, 3050, 3051, 3052, 3053, 3054, 3055, 3056, 3057, 3058, 3059, 3060, 3061, 3062, 3063, 3064, 3065, 3066, 3067, 3068, 3069, 3070, 3071, 3072, 3073, 3074, 3075, 3076, 3077, 3078, 3079, 3080, 3081, 3082, 3083, 3084, 3085, 3086, 3087, 3088, 3089, 3090, 3091, 3092, 3093, 3094, 3095, 3096, 3097, 3098, 3099, 3100, 3101, 3102, 3103, 3104, 3105, 3106, 3107, 3108, 3109, 3110, 3111, 3112, 3113, 3114, 3115, 3116, 3117, 3118, 3119, 3120, 3121, 3122, 3123, 3124, 3125, 3126,

[illegible]

Figure 1. Example of a 2D grid world environment. The grid is 10x10. The agent starts at (0,0) and moves to (1,0). The goal is at (9,9). The grid contains obstacles (black cells) and rewards (red cells). The agent's path is shown in blue.

| | | | | | | | | | | | | |
|-----------------------------------|--|--|--|--|--|--|--|--|--|--|--|--|
| Date: 2301 192304g Spc: 3304 Yolo | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |

[illegible]

| Port 1E, 00, ..., 255 (see Table 2.15.000A) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 | 128 | 129 | 130 | 131 | 132 | 133 | 134 | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 142 | 143 | 144 | 145 | 146 | 147 | 148 | 149 | 150 | 151 | 152 | 153 | 154 | 155 | 156 | 157 | 158 | 159 | 160 | 161 | 162 | 163 | 164 | 165 | 166 | 167 | 168 | 169 | 170 | 171 | 172 | 173 | 174 | 175 | 176 | 177 | 178 | 179 | 180 | 181 | 182 | 183 | 184 | 185 | 186 | 187 | 188 | 189 | 190 | 191 | 192 | 193 | 194 | 195 | 196 | 197 | 198 | 199 | 200 | 201 | 202 | 203 | 204 | 205 | 206 | 207 | 208 | 209 | 210 | 211 | 212 | 213 | 214 | 215 | |

[illegible]

Memory Mapping video

The current memory mapping is maintained by the hardware in eight MMUs with each MMU handling 4MB of MMIO. MMIOs holding an 8Kb size of MMIOs are mapped to the MMIOs. The MMIOs are mapped to the MMIOs. The MMIOs are mapped to the MMIOs. The MMIOs are mapped to the MMIOs.

the traditional parking methodism used in various specimen
 methods is to "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" "11" "12" "13" "14" "15" "16" "17" "18" "19" "20" "21" "22" "23" "24" "25" "26" "27" "28" "29" "30" "31" "32" "33" "34" "35" "36" "37" "38" "39" "40" "41" "42" "43" "44" "45" "46" "47" "48" "49" "50" "51" "52" "53" "54" "55" "56" "57" "58" "59" "60" "61" "62" "63" "64" "65" "66" "67" "68" "69" "70" "71" "72" "73" "74" "75" "76" "77" "78" "79" "80" "81" "82" "83" "84" "85" "86" "87" "88" "89" "90" "91" "92" "93" "94" "95" "96" "97" "98" "99" "100" "101" "102" "103" "104" "105" "106" "107" "108" "109" "110" "111" "112" "113" "114" "115" "116" "117" "118" "119" "120" "121" "122" "123" "124" "125" "126" "127" "128" "129" "130" "131" "132" "133" "134" "135" "136" "137" "138" "139" "140" "141" "142" "143" "144" "145" "146" "147" "148" "149" "150" "151" "152" "153" "154" "155" "156" "157" "158" "159" "160" "161" "162" "163" "164" "165" "166" "167" "168" "169" "170" "171" "172" "173" "174" "175" "176" "177" "178" "179" "180" "181" "182" "183" "184" "185" "186" "187" "188" "189" "190" "191" "192" "193" "194" "195" "196" "197" "198" "199" "200" "201" "202" "203" "204" "205" "206" "207" "208" "209" "210" "211" "212" "213" "214" "215" "216" "217" "218" "219" "220" "221" "222" "223" "224" "225" "226" "227" "228" "229" "230" "231" "232" "233" "234" "235" "236" "237" "238" "239" "240" "241" "242" "243" "244" "245" "246" "247" "248" "249" "250" "251" "252" "253" "254" "255" "256" "257" "258" "259" "260" "261" "262" "263" "264" "265" "266" "267" "268" "269" "270" "271" "272" "273" "274" "275" "276" "277" "278" "279" "280" "281" "282" "283" "284" "285" "286" "287" "288" "289" "290" "291" "292" "293" "294" "295" "296" "297" "298" "299" "300" "301" "302" "303" "304" "305" "306" "307" "308" "309" "310" "311" "312" "313" "314" "315" "316" "317" "318" "319" "320" "321" "322" "323" "324" "325" "326" "327" "328" "329" "330" "331" "332" "333" "334" "335" "336" "337" "338" "339" "340" "341" "342" "343" "344" "345" "346" "347" "348" "349" "350" "351" "352" "353" "354" "355" "356" "357" "358" "359" "360" "361" "362" "363" "364" "365" "366" "367" "368" "369" "370" "371" "372" "373" "374" "375" "376" "377" "378" "379" "380" "381" "382" "383" "384" "385" "386" "387" "388" "389" "390" "391" "392" "393" "394" "395" "396" "397" "398" "399" "400" "401" "402" "403" "404" "405" "406" "407" "408" "409" "410" "411" "412" "413" "414" "415" "416" "417" "418" "419" "420" "421" "422" "423" "424" "425" "426" "427" "428" "429" "430" "431" "432" "433" "434" "435" "436" "437" "438" "439" "440" "441" "442" "443" "444" "445" "446" "447" "448" "449" "450" "451" "452" "453" "454" "455" "456" "457" "458" "459" "460" "461" "462" "463" "464" "465" "466" "467" "468" "469" "470" "471" "472" "473" "474" "475" "476" "477" "478" "479" "480" "481" "482" "483" "484" "485" "486" "487" "488" "489" "490" "491" "492" "493" "494" "495" "496" "497" "498" "499" "500" "501" "502" "503" "504" "505" "506" "507" "508" "509" "510" "511" "512" "513" "514" "515" "516" "517" "518" "519" "520" "521" "522" "523" "524" "525" "526" "527" "528" "529" "530" "531" "532" "533" "534" "535" "536" "537" "538" "539" "540" "541" "542" "543" "544" "545" "546" "547" "548" "549" "550" "551" "552" "553" "554" "555" "556" "557" "558" "559" "560" "561" "562" "563" "564" "565" "566" "567" "568" "569" "570" "571" "572" "573" "574" "575" "576" "577" "578" "579" "580" "581" "582" "583" "584" "585" "586" "587" "588" "589" "590" "591" "592" "593" "594" "595" "596" "597" "598" "599" "600" "601" "602" "603" "604" "605" "606" "607" "608" "609" "610" "611" "612" "613" "614" "615" "616" "617" "618" "619" "620" "621" "622" "623" "624" "625" "626" "627" "628" "629" "630" "631" "632" "633" "634" "635" "636" "637" "638" "639" "640" "641" "642" "643" "644" "645" "646" "647" "648" "649" "650" "651" "652" "653" "654" "655" "656" "657" "658" "659" "660" "661" "662" "663" "664" "665" "666" "667" "668" "669" "670" "671" "672" "673" "674" "675" "676" "677" "678" "679" "680" "681" "682" "683" "684" "685" "686" "687" "688" "689" "690" "691" "692" "693" "694" "695" "696" "697" "698" "699" "700" "701" "702" "703" "704" "705" "706" "707" "708" "709" "710" "711" "712" "713" "714" "715" "716" "717" "718" "719" "720" "721" "722" "723" "724" "725" "726" "727" "728" "729" "730" "731" "732" "733" "734" "735" "736" "737" "738" "739" "740" "741" "742" "743" "744" "745" "746" "747" "748" "749" "750" "751" "752" "753" "754" "755" "756" "757" "758" "759" "760" "761" "762" "763" "764" "765" "766" "767" "768" "769" "770" "771" "772" "773" "774" "775" "776" "777" "778" "779" "780" "781" "782" "783" "784" "785" "786" "787" "788" "789" "790" "791" "792" "793" "794" "795" "796" "797" "798" "799" "800" "801" "802" "803" "804" "805" "806" "807" "808" "809" "810" "811" "812" "813" "814" "815" "816" "817" "818" "819" "820" "821" "822" "823" "824" "825" "826" "827" "828" "829" "830" "831" "832" "833" "834" "835" "836" "837" "

2. US Specimen used for FF on the 3 used are:
 1. 100% Specimen: 100% material, 100%
 2. 100% Specimen: 100% material, 100%
 3. 100% Specimen: 100% material, 100%

[illegible]

In the first section of code, we 7F0D and 1F0Dh are used as 2176 and 8192 bytes, respectively.

For CFFDh water, the % of available nitrogen in the water column is 0.22:0.00 g per kg, the total available nitrogen is 0.22 g per kg.

New's territory can be parced into five top "EK"

Also, the huge 70-day FFFHuge is still placed 8K below the 100th position, for example, by 40%.

Wavelengths: FFD_H, DFFCH₁, FFDH₂, FFF_H, Yellow 690 nm,
column 10° dike top, 6X is normal half-height mask & selected
the entire 5x6 images.

7. Further, the 5-bit mode word FF50 is extended so that bits D0-D6 contain the same number as bits D2-D9 in form a 4-bit word number of the mode: 8F.

in the 2K spectrum. Next, port 1FF0h continues to function as I/O module. Bank 0 of the 23 or 277h can be used to map 6K bank 7 or up to 4096 in the bottom 6K.

Values to ports 7FFCh, 0FFCh, 1FFCh and 2FFCh will be 0000 0000 0000 0000 (0) and the top 64 is 0000 0000 0000 0000 (0) made as side and 0000 0000 0000 0000 (0).

7-tetragon 1024K mode per L² 7h D2 = 0 enables 7-tetragon;
 1024K mapping and on FFF'h D2 = 0; 0x00000000 7X
 Spectrum mapping as per 000 above

At Fullgate, 024K mode active, port 7FFFh is unlocked and 05 (the lock bit) is reprogrammed as another bank bit.

*The first page of paper used on my ink is taken up by 7FFDh!
cde 08.07.08.02.0'.00' to reach 52 in 16K bytes

C. The size of the FFH control block is 16 bytes (1016H) and the HWT selector and 32-bit FFH control bits are 16 bytes (1016H) in size. The total size of the FFH control block is 32 bytes (2032H).

While in ports 7F3Dh, 0FFDh, 1FFDh and EFFFh will cause the bulletin 1FF7 and the top 10F to be output. If **ALLDATA** mode is selected, the entire 50k is output.

For 100% compatibility with the original banking methods listed above, users must use 5FDH, an algorithm designed to emulate the decisions with Neochelle 30-33 102h 85h.

The Expansion Bus

The Expansion Bus is found in the back of the ZX Spectrum Next and exposes its CPU to the world. As it too gets addressed by **IN** and **OUT** commands. It is listed below.

| AT | 2B | 2B | Reserved |
|----------|----|----|----------|
| A9 | 27 | 27 | A' 0 |
| BUSACK | 26 | 26 | A8 |
| MASTRBS | 25 | 25 | MASTRI |
| A4 | 24 | 24 | MA |
| A0 | 23 | 23 | MA |
| A0 | 22 | 22 | MA |
| A7 | 21 | 21 | MA |
| RESET | 20 | 20 | MA |
| BUSREQ | 19 | 0 | MA |
| N: | 18 | 8 | MA |
| N: | 17 | 7 | MA |
| Reserved | 16 | 6 | MA |
| MASTRBS | 15 | 5 | MA |
| GND | 14 | 4 | MA |
| MASTRBS | 13 | 3 | MA |
| A3 | 12 | 2 | MA |
| A2 | 1 | 1 | MA |
| A | 10 | 0 | MA |
| A0 | 9 | 0 | MA |
| CR | 8 | 8 | MA |
| A0 | 7 | 7 | MA |
| A0 | 6 | 6 | MA |
| Key | 5 | 5 | Key |
| MASTRBS | 4 | 4 | MASTRBS |
| A4 | 3 | 3 | MA |
| A' 2 | 2 | 2 | A' 3 |
| A' 4 | | | A' 5 |

The ZI Spectrum Next Expansion Bus

[illegible]

As an receiver the unregulated power from the 150 and 250 plug a higher voltage DSB final voltage will be approximately 150 and 250 volts respectively.

Chapter 23 The Memory

Overview

explore how your computer stores information we put into it

tion or we can do it ourselves as long as we know how

bytes at one time. Hold onto this information for now as it's important.

ROM and RAM

herent and read-only (see for example Chapter 7)

Next, indeed has ROM

The Memory Map

having it? And you would be absolutely right to ask this

cause the CPL is given a new window into a different bank in physical memory. This way the usable physical memory can far exceed the memory the CPU can normally see while at the same time older software is completely unaware and will continue to run properly without performing any bank switching.

Memory Management

There are two banking schemes employed in the ZX Spectrum Next: Standard and MMU-based banking. The Standard scheme is inherited from the +3 and the other 28K Spectrum models. The MMU scheme co-exists with the Standard scheme but it is unique to the ZX Spectrum Next.



Fig. 46 Standard (NextBASIC) memory map

As you can see in the memory map NextBASIC uses the available 64K of addressable memory is divided into four slots of 16K each with the bottom slot always occupied by ROM. Standard banking, inherited from prior Spectrum models, selects which 16K ROM is visible in the bottom 16K slot (addresses 0 to 16383) and which 16K RAM bank is visible in the top 16K slot (addresses 49152 to 65535).

The Spectrum 3 introduced a new so-called **allRAM** mode that could place a limited selection of arrangements of four 16K RAM banks into all four slots. This was not widely used and is often forgotten by programmers who mostly argue the 28K Spectrum models prior to the +3. A good example of **allRAM** mode is running CP/M that requires RAM at the bottom of the address map.

There is a total of four 16K ROMs to select from (inherited from the +3) and a total of 48 16K RAM banks available (12 in 2048K ZX Spectrum Nexts). If you make a quick calculation that accounts for 832K in the unexpanded issue 2 ZX Spectrum Next, the remaining portion of the 1024K is allocated to other uses, most notably to divMMC memory. The NextZXOS Startup menu reports available RAM only, which will be either 768K or 1792K.

The Standard banking scheme is controlled by hardware I/O ports (covered in the previous chapter) and via the **BANK** command and its variants which we will examine soon.

The MMU (memory management unit) scheme is diagrammed below. It is much more flexible in that it can map any 8K bank of physical RAM into any 8K slot of the CPU's addressable memory.

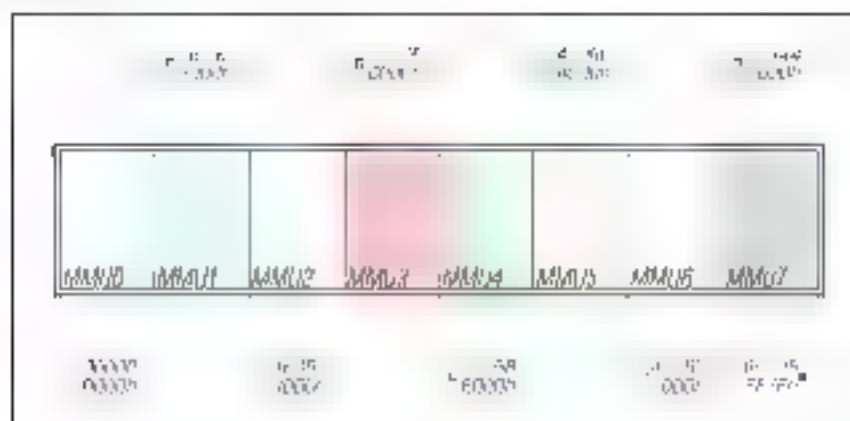


Fig. 47 MMU-based memory map

Physical memory is divided into 64 banks of 8K each. MMIO and I/O MMIO and physical memory is broken into 4096 banks. Finding a specific 8K bank in the address space is a little bit of a pain, so we might say that Bank *n* has been written to MMIO.

Since NextBASIC exposes physical memory banks using the Standard scheme's 64 slots, we'll concentrate only on this. Much information on using the ZX Spectrum Next's MMIO system can be found at the end of this chapter in Internet resources such as the Spectrum Next Wiki, wiki.spectrumnext.org and in volume 2, *Advanced ZX Spectrum Next programming* of this manual.

Reading and Writing to Memory

The initial release of Spectrum Next, *NextZXNS* and *NextBASIC*, had read and write memory in your hands! As it has been demonstrated in previous chapters, we sometimes need to examine the memory's contents and to modify them. For those cases, NextBASIC provides a series of commands and functions to examine and modify memory, both in the memory map as well as in the whole of the physical memory. These are all variations of two main keywords, namely, the **PEEK** and **PEEK\$** functions to read the contents of memory and the **POKE** command (to alter the contents of memory). The full list follows:

| Command | Description |
|---|---|
| PEEK <i>addr</i> | Reads a 16-bit value at <i>addr</i> . |
| POKE <i>addr</i> , <i>v</i> | Changes the contents of address <i>addr</i> to the 16-bit value <i>v</i> . |
| DPEEK <i>addr</i> , <i>o</i> | Reads a 16-bit value at <i>addr</i> + <i>o</i> in bank <i>addr</i> / 4096. |
| DPOKE <i>addr</i> , <i>v</i> | Changes the contents of addresses starting at <i>addr</i> / 4096, <i>addr</i> / 256 to contain the 16-bit value <i>v</i> . |
| PEEK\$ <i>addr</i> , <i>len</i> | Reads <i>len</i> bytes starting at <i>addr</i> and stores the addresses up to <i>len</i> bytes at <i>addr</i> + 1 in <i>addr</i> + 2. <i>len</i> must be a multiple of 2. The string is null-terminated. |
| POKE <i>addr</i> , <i>s</i> | Writes a string <i>s</i> to the addresses beginning with <i>addr</i> . |
| BANK <i>n</i> , PEEK <i>o</i> | Reads a 16-bit value at <i>o</i> in bank <i>n</i> . |
| BANK <i>n</i> , POKE <i>o</i> , <i>v</i> | Changes the contents in bank <i>n</i> at offset <i>o</i> to value <i>v</i> . |
| BANK <i>n</i> , DPEEK <i>o</i> | Reads the word stored in bank <i>n</i> at offset <i>o</i> to <i>o</i> + 1. |
| BANK <i>n</i> , DPOKE <i>o</i> , <i>v</i> | Changes the contents of bank <i>n</i> starting at offset <i>o</i> (0, <i>o</i> + 1) to contain the 16-bit value <i>v</i> . |
| BANK <i>n</i> , PEEK\$ <i>len</i> , <i>addr</i> | Reads <i>len</i> bytes starting at <i>addr</i> in bank <i>n</i> and stores the addresses up to <i>len</i> bytes at <i>addr</i> + 1 in <i>addr</i> + 2. <i>len</i> must be a multiple of 2. The string is null-terminated. |
| BANK <i>n</i> , POKE <i>o</i> , <i>s</i> | Writes a string <i>s</i> in bank <i>n</i> beginning at offset <i>o</i> . |

Table 22: PEEK and POKE variants

As you can see from the table above, NextBASIC provides us with a wealth of options to manipulate the contents of both the bank memory map and the physical memory as a whole. These are complemented by the extended options provided by the **BANK** command, which we will examine further below, and cover all necessary memory manipulation that may arise in the course of writing a program.

Before we continue further with examining the **PEEK**, **PEEK\$** and **POKE** keywords, however, with a warning of sorts: usage of the main **BANK** variants is extremely discouraged. In short, "it's bad" – you should never use the **BANK** variants at all if it's possible. The case for this will be explained and goes back to Memory Banking.

Let's explain, as we saw earlier, NextZXNS and NextBASIC update portions of the memory map like the system variables, in order to display memory, need be. What this means is that you can't really be sure a value you **POKE**d into the memory map will be there when you try to recover it with **PEEK**, unless you take some measures first!

Furthermore, **POKE**ing into the memory map, unless you absolutely know what you're doing, can have multiple disastrous effects with it. It's best to avoid it as much as possible.

We'll first give an example of what could go wrong (it's fortunately safe as an example) and then we'll take a detour and explain how the memory map itself is organised from a NextBASIC perspective before returning to PEEK, POKE and their variants. Type

```
10 POKE 16384,"ABCabc"
20 CLS a%=PEEK# (16384,6)
40 PRINT a%
```

From what we've talked about thus far, the intention of the program is obvious. (or now also never mind what line 1 does, we'll discuss it later). First we put the word ABCabc into address 16384 of the memory map. Then we try to extract it from the same memory location. RUN the program. What do you see? Certainly not ABCabc you were expecting. Now modify lines 20 and 30 and replace 16384 with 20000 in both lines and RUN the program again.

This is perhaps a contrived example but it shows what happens when you try to use memory that is also being used by something else. In this case, address 16384 is where the contents of the display is stored. After placing the string with POKE in address 16384, a CLS is executed which clears the display and the stored string at the same time.

Here is a trickier example

```
10 LAYER 1 2
20 POKE 16384,255
30 POKE 24576,255
40 PRINT AT 1,0,"16384 = ",
    PEEK 16384
50 PRINT "24576 = "; PEEK
    24576
```

This program selects Layer 1 2 (HiRes) and then creates two solid and adjacent character sized lines into the display via the POKE commands in lines 20 and 30. Running the program the results almost seem correct except the character sized line is only one character wide.

The POKE to 24576 did not go to the display in bank 5 because NextBASIC placed a different memory page in the memory map to cover the last half of bank 5.

Contrast with the following program that does all its PEEKs and POKEs to bank 5 (the BANK commands will be explained in more detail later). As we will see, PEEK and POKE into a 16K bank is done using an offset into said bank. This means that the address range is 0 through 16383. Banks are only 16K long after all. Bank 5, which holds the display, is normally placed at address 16384 in the memory map. Performing therefore a POKE into address 16384 is the same as POKE to offset 0 in bank 5. Likewise address 24576 corresponds to offset 8192 in bank 5.

```
10 LAYER 1 2
20 BANK 5 POKE 0,255
30 BANK 5 POKE 8192,255
40 PRINT AT 1,0,"16384 = ",%
    BANK 5 PEEK 0
50 PRINT "24576 = ",% BANK 5
    PEEK 8192
```

This time the POKE to 24576 (offset 8192) does go to bank 5 and you will see the solid line twice as wide as the first program.

NextZXOS and NextBASIC memory allocation

Before we begin to elaborate on NextZXOS' memory usage, it should be mentioned that Standard memory management and MMU management are internally synchronised, or most cases. Every time a 16K bank is being paged in, the equivalent MMU unit gets the 8K bank component of the larger 16K bank NextZXOS uses. As mentioned previously, NextZXOS also supports **allRAM** mode where the ROM is paged out (this is mainly used by CP/M). With this information out of the way, let's see how NextZXOS uses the memory.

By default, the first 9 RAM banks are used as follows:

| Bank | Description | Address Range |
|------|--|---------------|
| 0 | Standard 48K Spectrum memory | 48-52 00000 |
| 1 | RAMdisk | |
| 2 | Standard 48K Spectrum memory | 32768-49-51 |
| 3 | RAMdisk | |
| 4 | RAMdisk | |
| 5 | Standard 48K Spectrum memory | 16384-32767 |
| 6 | RAMdisk | |
| 7 | Used for workspace and data structures by NextZXOS | |
| 8 | Used for additional screen data (for CoFies, MFiles and MComport and other data by NextZXOS) | |
| 9-11 | Available for user programs (By default, banks 9-10 and 11 are used by Layer 2) | |

Generally speaking, banks 9+ are always available to the programmer and can be accessed using the **BANK** command, while banks 0-8 can be used with the following exceptions:

- Bank 0 can be used, only if **CLEAR** has set the **RAMTOP** to below 48152
- Bank 2 can be used, only if **CLEAR** has set the **RAMTOP** to below 32768
- Banks 1,3,4,6 can be used if the **BANK 1346 USR** command has been used
- Banks 7 and 8 can never be used
- Bank 5 can be used with caution
- Banks 9-10 and 11 can be used for other purposes if you aren't using Layer 2 or you have changed their assignments with the **LAYER BANK** command

From the above, it is easy to surmise what the initial bank assignments are after boot:

| Slot 1 | Slot 2 | Slot 3 | Slot 4 |
|--------|--------|--------|--------|
| ROM | Bank 5 | Bank 2 | Bank 0 |

In case you were thinking that this looks easy enough, you could page in any bank, wait, don't, in actuality, NextZXOS and NextBASIC expect certain things to be in certain places at all times within the memory map which is organised in the following manner:

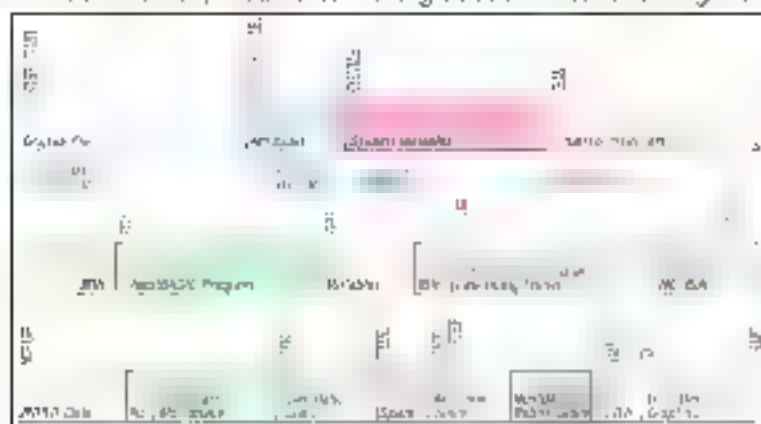


Fig. 48 Memory map usage by NextBASIC

As you can see, the memory map is a very complex structure. It is a map of the memory, showing the location of various areas and their size. The memory map is a very important part of the system, and it is used by the operating system to manage the memory. The memory map is a very complex structure, and it is used by the operating system to manage the memory. The memory map is a very important part of the system, and it is used by the operating system to manage the memory.

see why

The memory map is a very complex structure. It is a map of the memory, showing the location of various areas and their size. The memory map is a very important part of the system, and it is used by the operating system to manage the memory.

appearing as needed and managed by NextZKOS

The memory map is a very complex structure. It is a map of the memory, showing the location of various areas and their size. The memory map is a very important part of the system, and it is used by the operating system to manage the memory. The memory map is a very complex structure, and it is used by the operating system to manage the memory.

means nothing to NextBASIC

The memory map is a very complex structure. It is a map of the memory, showing the location of various areas and their size. The memory map is a very important part of the system, and it is used by the operating system to manage the memory. The memory map is a very complex structure, and it is used by the operating system to manage the memory.

Memory Areas and their use

It is important to generally know how things are laid out in the memory map.

The memory map is a very complex structure. It is a map of the memory, showing the location of various areas and their size. The memory map is a very important part of the system, and it is used by the operating system to manage the memory. The memory map is a very complex structure, and it is used by the operating system to manage the memory.

The memory map is a very complex structure. It is a map of the memory, showing the location of various areas and their size. The memory map is a very important part of the system, and it is used by the operating system to manage the memory. The memory map is a very complex structure, and it is used by the operating system to manage the memory.

The memory map is a very complex structure. It is a map of the memory, showing the location of various areas and their size. The memory map is a very important part of the system, and it is used by the operating system to manage the memory. The memory map is a very complex structure, and it is used by the operating system to manage the memory.

The memory map is a very complex structure. It is a map of the memory, showing the location of various areas and their size. The memory map is a very important part of the system, and it is used by the operating system to manage the memory. The memory map is a very complex structure, and it is used by the operating system to manage the memory.

The memory map is a very complex structure. It is a map of the memory, showing the location of various areas and their size. The memory map is a very important part of the system, and it is used by the operating system to manage the memory. The memory map is a very complex structure, and it is used by the operating system to manage the memory.

The *Spare* area contains the space so far unused

The *Machine Stack* area is space reserved for the CPL stack

Similarly, the *NextBASIC* return stack area which was mentioned in Chapter 4 maintains a record of your program's currently active subroutine and procedure calls, loops and error handlers.

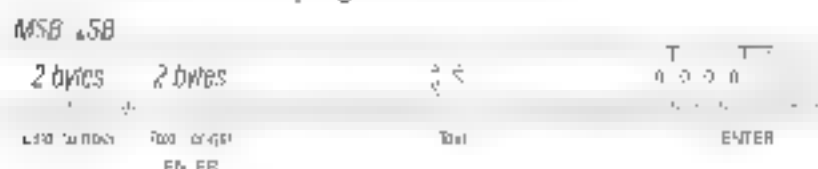
The byte pointed by the `RAMTOP` variable shows the maximum address that is reserved for use by a *NextBASIC* program. We will visit this in more detail in the section about the `CLEAR` command below.

Finally, the *User Defined Graphics* area holds all the definitions for the system's `UDGs` as discussed in Chapter 13.

NextBASIC Data Structures

NextBASIC stores numbers, strings, arrays, programming lines and `FOR...NEXT` loops in strictly defined forms called *data structures*. The following discuss all these data structures that are user accessible. Integer-based variables, arrays and control structures are not available to the user and are hidden by *NextZXOS* in protected memory areas so they're not covered here.

Each line of *NextBASIC* program has the form



Note that, in contrast with all other cases of two-byte numbers in the *Z80* (the line number here), is stored with its more significant byte 'MSB' first; that is to say, in the order that you write them down (also known as *Big-Endian* order).

A numerical constant in the program appears as ASCII text followed by its binary form, using the character `CHR$ 14` followed by five bytes for the number itself.

The variables have different formats according to their features. The letters in the names should be thought as starting off in lower case. The available variants and their formats are:

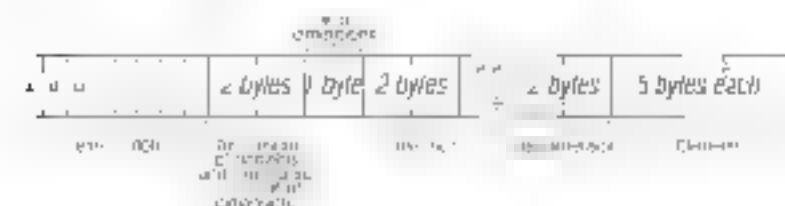
Number whose name is one letter only



Number whose name is longer than one letter



Array of numbers



Array of numbers whose name is longer than one letter



Specifically for arrays the order of the elements is as follows

- first the elements for which the first subscript is 1
- next the elements for which the first subscript is 2
- next the elements for which the first subscript is 3

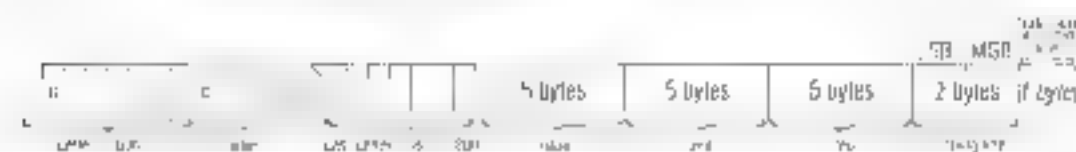
and so on for all possible values of the first subscript

The elements with a given first subscript are ordered in the same way using the second subscript and so on down to the last. As an example, the elements of the 3 x 6 array `b` in Chapter 11 are stored in the order `b(1,1)`, `b(1,2)`, `b(1,3)`, `b(1,4)`, `b(1,5)`, `b(1,6)`, `b(2,1)`, `b(2,2)`, `b(2,3)`, `b(2,4)`, `b(2,5)`, `b(2,6)`, `b(3,1)`, `b(3,2)`, `b(3,3)`, `b(3,4)`, `b(3,5)`, `b(3,6)`.

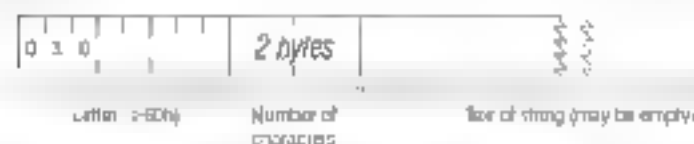
Control variable of a **FOR...NEXT** loop



Control variable of a **FOR...NEXT** loop whose name is longer than one letter



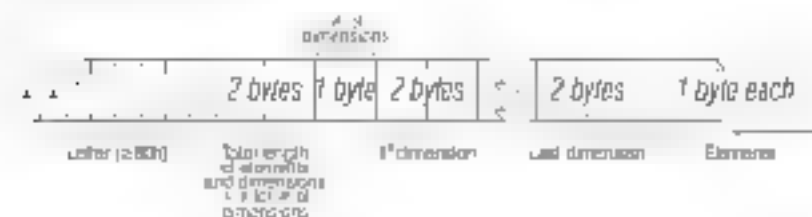
String



String whose name is longer than one letter



Array of characters



Array of characters whose name is longer than one letter

| | byte | byte | bytes | bytes | characters |
|-----|------|------|-------|-------|------------|
| 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 | 1 | 1 |
| 3 | 1 | 1 | 1 | 1 | 1 |
| 4 | 1 | 1 | 1 | 1 | 1 |
| 5 | 1 | 1 | 1 | 1 | 1 |
| 6 | 1 | 1 | 1 | 1 | 1 |
| 7 | 1 | 1 | 1 | 1 | 1 |
| 8 | 1 | 1 | 1 | 1 | 1 |
| 9 | 1 | 1 | 1 | 1 | 1 |
| 10 | 1 | 1 | 1 | 1 | 1 |
| 11 | 1 | 1 | 1 | 1 | 1 |
| 12 | 1 | 1 | 1 | 1 | 1 |
| 13 | 1 | 1 | 1 | 1 | 1 |
| 14 | 1 | 1 | 1 | 1 | 1 |
| 15 | 1 | 1 | 1 | 1 | 1 |
| 16 | 1 | 1 | 1 | 1 | 1 |
| 17 | 1 | 1 | 1 | 1 | 1 |
| 18 | 1 | 1 | 1 | 1 | 1 |
| 19 | 1 | 1 | 1 | 1 | 1 |
| 20 | 1 | 1 | 1 | 1 | 1 |
| 21 | 1 | 1 | 1 | 1 | 1 |
| 22 | 1 | 1 | 1 | 1 | 1 |
| 23 | 1 | 1 | 1 | 1 | 1 |
| 24 | 1 | 1 | 1 | 1 | 1 |
| 25 | 1 | 1 | 1 | 1 | 1 |
| 26 | 1 | 1 | 1 | 1 | 1 |
| 27 | 1 | 1 | 1 | 1 | 1 |
| 28 | 1 | 1 | 1 | 1 | 1 |
| 29 | 1 | 1 | 1 | 1 | 1 |
| 30 | 1 | 1 | 1 | 1 | 1 |
| 31 | 1 | 1 | 1 | 1 | 1 |
| 32 | 1 | 1 | 1 | 1 | 1 |
| 33 | 1 | 1 | 1 | 1 | 1 |
| 34 | 1 | 1 | 1 | 1 | 1 |
| 35 | 1 | 1 | 1 | 1 | 1 |
| 36 | 1 | 1 | 1 | 1 | 1 |
| 37 | 1 | 1 | 1 | 1 | 1 |
| 38 | 1 | 1 | 1 | 1 | 1 |
| 39 | 1 | 1 | 1 | 1 | 1 |
| 40 | 1 | 1 | 1 | 1 | 1 |
| 41 | 1 | 1 | 1 | 1 | 1 |
| 42 | 1 | 1 | 1 | 1 | 1 |
| 43 | 1 | 1 | 1 | 1 | 1 |
| 44 | 1 | 1 | 1 | 1 | 1 |
| 45 | 1 | 1 | 1 | 1 | 1 |
| 46 | 1 | 1 | 1 | 1 | 1 |
| 47 | 1 | 1 | 1 | 1 | 1 |
| 48 | 1 | 1 | 1 | 1 | 1 |
| 49 | 1 | 1 | 1 | 1 | 1 |
| 50 | 1 | 1 | 1 | 1 | 1 |
| 51 | 1 | 1 | 1 | 1 | 1 |
| 52 | 1 | 1 | 1 | 1 | 1 |
| 53 | 1 | 1 | 1 | 1 | 1 |
| 54 | 1 | 1 | 1 | 1 | 1 |
| 55 | 1 | 1 | 1 | 1 | 1 |
| 56 | 1 | 1 | 1 | 1 | 1 |
| 57 | 1 | 1 | 1 | 1 | 1 |
| 58 | 1 | 1 | 1 | 1 | 1 |
| 59 | 1 | 1 | 1 | 1 | 1 |
| 60 | 1 | 1 | 1 | 1 | 1 |
| 61 | 1 | 1 | 1 | 1 | 1 |
| 62 | 1 | 1 | 1 | 1 | 1 |
| 63 | 1 | 1 | 1 | 1 | 1 |
| 64 | 1 | 1 | 1 | 1 | 1 |
| 65 | 1 | 1 | 1 | 1 | 1 |
| 66 | 1 | 1 | 1 | 1 | 1 |
| 67 | 1 | 1 | 1 | 1 | 1 |
| 68 | 1 | 1 | 1 | 1 | 1 |
| 69 | 1 | 1 | 1 | 1 | 1 |
| 70 | 1 | 1 | 1 | 1 | 1 |
| 71 | 1 | 1 | 1 | 1 | 1 |
| 72 | 1 | 1 | 1 | 1 | 1 |
| 73 | 1 | 1 | 1 | 1 | 1 |
| 74 | 1 | 1 | 1 | 1 | 1 |
| 75 | 1 | 1 | 1 | 1 | 1 |
| 76 | 1 | 1 | 1 | 1 | 1 |
| 77 | 1 | 1 | 1 | 1 | 1 |
| 78 | 1 | 1 | 1 | 1 | 1 |
| 79 | 1 | 1 | 1 | 1 | 1 |
| 80 | 1 | 1 | 1 | 1 | 1 |
| 81 | 1 | 1 | 1 | 1 | 1 |
| 82 | 1 | 1 | 1 | 1 | 1 |
| 83 | 1 | 1 | 1 | 1 | 1 |
| 84 | 1 | 1 | 1 | 1 | 1 |
| 85 | 1 | 1 | 1 | 1 | 1 |
| 86 | 1 | 1 | 1 | 1 | 1 |
| 87 | 1 | 1 | 1 | 1 | 1 |
| 88 | 1 | 1 | 1 | 1 | 1 |
| 89 | 1 | 1 | 1 | 1 | 1 |
| 90 | 1 | 1 | 1 | 1 | 1 |
| 91 | 1 | 1 | 1 | 1 | 1 |
| 92 | 1 | 1 | 1 | 1 | 1 |
| 93 | 1 | 1 | 1 | 1 | 1 |
| 94 | 1 | 1 | 1 | 1 | 1 |
| 95 | 1 | 1 | 1 | 1 | 1 |
| 96 | 1 | 1 | 1 | 1 | 1 |
| 97 | 1 | 1 | 1 | 1 | 1 |
| 98 | 1 | 1 | 1 | 1 | 1 |
| 99 | 1 | 1 | 1 | 1 | 1 |
| 100 | 1 | 1 | 1 | 1 | 1 |

As you saw in the examples above, numerical values are represented as 5 bytes. These are *floating-point values*. In comparison, integers which are, as discussed in Chapter 6 and referenced in Chapters 8 through 11, as a *fixed 16 bit* (or two-byte, size) *floating-point* numbers can represent both decimal and integer values. Due to the calculations involved, their usage will slow down your programs, so avoid using them if you do not need decimal points or values higher than 65535.

For *floating-point* values, any number (except 0) can be written uniquely as $\pm m \times 2^e$

where \pm is the sign, m is the mantissa, which lies between $1/2$ and 1 (it cannot be 1), and e is a *biased exponent*.

Suppose you write the fractional m in binary. Because it is a fraction, it will have a binary point (like the decimal point in decimal) and then a binary fraction (like a decimal fraction). So in binary

| | | |
|----------------|---------------|-----------------------|
| one half | is written as | 1 |
| one quarter | is written as | 01 |
| three quarters | is written as | 11 |
| one tenth | is written as | 00011001 001100110011 |

and so on.

With our number m , because it is less than 1, there are no bits before the binary point, and because it is at least $1/2$, the bit immediately after the binary point is a 1. To store the number in the computer, we use *five bytes*, as follows:

write the first *eight* bits of the mantissa in the *second* byte (we know that the first bit is 1), the *second eight* bits in the *third* byte, the *third eight* bits in the *fourth* byte and the *fourth eight* bits in the *fifth* byte.

replace the first bit in the second byte which we know is 1 by the sign, 0 for plus, 1 for minus.

write the *exponent* +128 in the first byte.

For instance, suppose our number is $1/10$.

$$1/10 = 1/8 \times 2^3$$

Thus the mantissa m is 1100110011001100110011001100 in binary (since the 33rd bit is 1, we shall round the 32nd up from 0 to 1), and the exponent e is 3.

Applying our three rules gives the *five bytes*:

There is an alternate way of storing whole numbers between -65535 and +65535:

the *first* byte is 0

the *second* byte is 0 for a positive number, FFh for a negative one

the *third* and *fourth* bytes are the less and more significant bytes of the number (or the number +131072 if it is negative;

the *fifth* byte is 0

This is essentially the two's complement representation we discussed in Chapter 6 for integers with two extra bytes: one before and one after the number and an entire byte dedicated to the sign as opposed to one bit only. Compared to the integer type supplied by the Integer Expressions evaluator it's wasteful memory-wise and slower to process.

PEEK, POKE and their variants

Now that we've examined more thoroughly what the memory map looks like to *Nex(BASIC)* it's time to revisit the commands and functions that read and modify its contents.

To inspect the contents of one or more memory locations, we use the **PEEK**, **DPEEK** and **PEEK\$()** functions. The **PEEK** variant functions are always safe to use as they change nothing in memory; they can however give unpredictable results in cases where a memory location is marked for moving. As we saw however, there are places in memory which are unmovable: reading in the System variables area for example is always a predictable scenario. For instance, this program prints out the first 21 bytes in ROM (and their addresses):

```
10 PRINT 'Address', TAB 8, 'Byte'
20 FOR a=0 TO 20
30 PRINT a, TAB 8, PEEK a
40 NEXT a
```

All these bytes will probably be quite meaningless to you, but the processor understands them to be instructions telling it what to do.

DPEEK is similar but since it returns 16 bit values, the example above would have to be rewritten as follows:

```
10 PRINT 'Address'; TAB 8, 'Word'
20 FOR %a=0 TO 20 STEP 2
30 PRINT %a, TAB 8, DPEEK %a
40 NEXT %a
```

Generally speaking, (D)PEEKing into ROM is very much useless and it's much more likely that you'll use (D)PEEK to either read a system variable or read a value you've previously **POKEd** (D)PEEK\$() on the other hand returns the values at an address in memory in the form of a string. Its syntax is as follows:

PEEK\$(address, argument)

where *address* is any address in the memory map, while *argument* can be one of the following:

- 1 A number signifying a length of characters to be retrieved
- 2 A single tilde ~ character to find any bit-7 terminated string (that means that bit 7 of the last character in the string is set)
- 3 A tilde ~ character followed by the ASCII code of one character that terminates the string

Let's look at an example which helps us search in memory (albeit very slowly):

```
10 RJN AT 3 REM this takes a
   long time
20 FOR %a=0 TO 65535
30 PRINT AT 0,0,"Now scanning
   address ' %a
40 %b= PEEK$ (%a,8)
```



```

50 IF a$='Variable' THEN PRINT
   AT 1,0,'Found word at
   address  " %a: GO TO 70  REM
   stop iterating here and go
   below
60 NEXT %a
70 FOR %a=0 TO 65535
80 PRINT AT 3,0: 'Now scanning
   address  " %a
90 a$ = PEEK$ (%a, "101)
100 IF a$='Variable' THEN PRINT AT
    4,0, 'Found word at address  " %a
110 NEXT %a

```

You'll undoubtedly notice that line 70 says `Variable` instead of `Variable` in that, because the letter `a` or character we set `peek` + `in` look for is not included in the string returned by `PEEK$`. What this program actually finds is the address in the memory map where the 512's start, the separator, which is exactly the lines 70. It is very much pointless but was made to show the flexibility of `PEEK$()` as a binary alternative to a character search.

Normally this is much more likely to lead to NUL (null) terminated strings — 0 (FFh) terminated strings — 255 (FFh) terminated strings and perhaps CR terminated strings — 13 (0Dh) or other printable characters terminated strings as seen `PRINT`ed with line separator.

To change the contents of a RAM address in the memory map we use the `POKE` or `DPOKE` statements. These have the form

```

POKE address, value1, value2[, value3 [, valueN]]
DPOKE address, value1, value2, value3, value4

```

The ability in `POKE` gives you immense power over the computer. You know how to wield it and in most situations you pass it on to your user. It is very easy to poke the wrong value in the wrong address. Those was programs that took you into a cycle of error. Unfortunately you won't do the computer any permanent damage.

As we mentioned earlier, `POKE` is generally not safe to use with the computer's memory map unless you can know what you're doing. If the area you're modifying is locked (like say, the kernel's stack) or it's a system variable, the value will always be the same. It's also safe to `POKE` within the memory map if you have used the `CLEAR` command and modify the area above it.

Let's try modifying a system variable to show how powerful `POKE`ing can be.

Fun type test The first time you type `ENTER` you will hear a faint buzz and a buzzing sound. The variable that holds the length of this buzz is called `BAZ` and it's located in address `23608` (`5C38h`) within the System Variables area.

Now let's see how far we adjust that buzz. We'll start by looking what's its current value with

```
PRINT PEEK 23608
```

Then modify it with

```
POKE 23608, 15
```


type test again and press **ENTER**. The buzz indicating the error in your code showed in length. You can experiment with different values. The new value you enter must be between -255 and +255, and if it is negative then 256 is added to it.

POKE is not confined into a simple byte-sized value as you may have surmised. In fact, it can accept a mix of numbers and strings in a comma-separated list of values with each accepting an optional hide or character suffix in the case of numeric values. The optional hide suffix after each value makes that value 6 bits wide (a word) while in the case of strings, the optional hide suffix sets the most significant bit of the last character in the string (usually known as bit7-termination). This is sometimes used in order to store variable-length strings in a compact way. However, it's usually more convenient to use a byte such as 0 (NUL) null-termination, 255 (FFh) or 13 (0Ch, 'CR') to terminate a variable-length string in memory. Note that for **DPOKE**, the hide is used after numeric values to specify values that should be written to memory as a byte rather than a 16-bit word. You can therefore think of the hide as a *write this value in the opposite way to the default for this command designator*. Let's see a few examples.

```
POKE 32768,200
```

modifies the contents of the byte address 32768 to 200

```
POKE 32768,8,9,10 "test",30000",55
```

modifies the contents of address 32768 to 8, address 32769 to 9, address 32770 to 10, addresses 32771 to 32774 to contain the string **test** (or in other words the values 116, 101, 115 and 116 respectively, the ASCII codes for the letters making up the word **test**), addresses 32775 and 32776 to contain values 48, 117 respectively (or $17 \times 256 + 48 = 30000$) and finally address 32777 to 55. In other words we **POKEd** 3 bytes, a string, a word and a byte.

```
DPOKE 32768,1000,2000 3000,100",2
```

modifies addresses 32768 through 32775, pokes 3 words (a byte, a byte and a word)

In Chapter 13 we briefly discussed **POKE USR "letter"**. That may look like a separate variant of **POKE** but in reality **USR "letter"** is just as shorthand to the address of the JDG defined by letter. Here is a small caveat that when used in a single value context, 8 successive **POKE USR** commands must be given, one for each row in the 8x8 matrix of the JDG, so it's always better if we use it in a list of values context, like so:

```
POKE USR "A",1,3,7,15,31,63,127,255
```

which redefines JDG A.

Using **POKE** with strings is equally powerful so it deserves a separate example. Let's use the example that used **PEEK\$()** to search for a string in code. In demonstration a bit of NextBASIC memory areas magic. First delete all lines after 60 and modify line 20 to read:

```
20 %a=22000 TO 55535
```

(This change is to make sure the program doesn't take forever)

RUN the program and when you find the address, note it down, then do the following:

```
POKE address, 'Horrible'
```

where **address** is the address you noted earlier. Press **ENTER**, then write **LIST** and look at line 60. See? Magic.

Note that using the first form of **POKE** to any address between 0 and 16383 (the ROM slot) will have no effect regardless of what you attempt to do as shown by this example:

```
FOR %f=0 TO 16383 POKE %f,0 NEXT %f
```


The same, however, is entirely accurate in **DPOKE** and the similar **POKE** version of the command. For example, both the commands that allow will NOT write in the ROM slot but WILL write in the RAM slot, so happens as you see from the previous figure, that the test area right after the ROM is filled with FFFF, so you'll see a visual test immediately.

```
POKE 16383, 'This is a test.' PAUSE 0
```

and

```
DPOKE 16383, 65535 PAUSE 0
```

will not produce a visible result in the upper left corner of the display while the ROM slot is not affected.

CLEAR

When looking at the different memory areas maintained by NextBASIC, we have mentioned the system variable **RAMTOP**. This variable, located at address 23730, contains the address of the last byte used by NextBASIC. Even **NEW**, which clears the RAM, only tries so far as this address, so it doesn't change the user-defined graphics. You can change the address **RAMTOP** points to by putting it as an numeric argument in a **CLEAR** statement as follows:

```
CLEAR new RAMTOP
```

This effectively does 4 things:

- clears out all the variables
- clears the display file (like **CLS**)
- does **RESTORE**
- clears the NextBASIC return stack and puts it at the new **RAMTOP** address, assuming that this lies between the logical stack and the physical end of RAM, otherwise it leaves **RAMTOP** as it was.

RUN also performs a **CLEAR**, although it never changes **RAMTOP**.

Using **CLEAR** in this way, you can either move **RAMTOP** up (to take more room for NextBASIC) by overwriting the user-defined graphics, or you can move it down to take more RAM than is physically free. **NEW** can also be used to ensure that the machine's stack is below **BFE0h:40120** when entering a call NextZX05. This means that the stack will not have to be subsequently moved within your own machine code.

Type **NEW** select NextBASIC, then **CLEAR 23800** to get some idea of what happens to the machine when it fills up. You'll immediately get an **M RAMTOP** no good error message. Typing **CLEAR 23900** will return **OK** to a prompt, while a further **CLEAR** will stop with a buzzing sound very quickly. That means that the NextBASIC user program memory is now full and you will have to make room before typing any more. There are also warning messages with roughly the same meaning: **4 Out of memory** and **G No room for line**.

It's worth mentioning that the Clear option in the NextBASIC menu, accessible by pressing the **EDIT** key, can also be used to **CLEAR** memory and, depending on use, you have cleared **RAMTOP** or low and no longer have enough memory. Enter NextBASIC command **Is as with the extended drive**. It sets **RAMTOP** to the address of the current 16K area, the equivalent of **CLEAR % DPEEK 23675-1** (the less than the value in the **ICG** sysvar).

Memory Bank management with BANK

Under NextBASIC the system's memory capacity is shown in the on-screen menu, can also be queried programmatically by examining the new system variable **MAXBANK**, which contains the number of the highest usable bank in the system, normally 47 or 111.

3 The documentation incorrectly refers to memory information although measured in 8K banks.

To make all the extra memory easily accessible to the user, *NextBASIC* provides a special command called **BANK** which can be combined with a number of normal commands to extend their functionality to the whole of the ZX Spectrum Next's memory and not just the memory map addresses. We've seen some already used in the course of this guide, especially in chapters 14 through 17 as well as Chapter 20.

Memory banks are marked as *in-use* or *free* by the user or by commands that access them (**BANK**, **PEEK**, **PEEK\$**, **POKE**, **COPY**, **ERASE**, **USR**, **LAYER**, **LAYER**, **BANK** and **LOAD**, **BANK**). Users can mark a bank as *in-use* or as *free* by either using an explicit command from the list above or one of the two special commands **BANK NEW var** and **BANK n CLEAR**.

BANK NEW var

Reserves the next available free bank number and assigns it to the numeric variable *var* ready for use with and by other **BANK** commands. This command is useful for allocating banks for use in *NextBASIC*, allowing for cases where a resident machine code program has previously allocated banks for its own use.

Note that it is not essential to use this command, as commands such as **LOAD**, **BANK** will automatically allocate the specified bank for use by *NextBASIC*, but only if the specified bank is not already in use by a resident machine code program.

Let's try a small example. Assuming you have a 2048k ZX Spectrum Next, type the following program:

```
10 FOR %f = 0 TO 111
20 BANK NEW a
30 PRINT AT 0,0, "Allocating
   bank ", a
40 NEXT %f
```

Once you **RUN** it, the program will begin to allocate memory banks and print the ones it allocates; you'll notice two things. Allocation begins at bank 111 (47 using an unexpanded KS1/Issue 2 ZX Spectrum Next) and progresses backwards and that program execution will stop abruptly with a 4 Out of memory error report once you reach a bank that's allocated by the system as described in the *NextZXS and NextBASIC Memory Allocation* section. Indeed, if you use the dot command *mem* then you'll see that you have 0 banks free (0k). In order to free up a bank to be used, you will need to use the **BANK n CLEAR** command whose syntax is as follows:

BANK n CLEAR

Marks bank *n* as free for use by other parts of the system (eg dot commands).

Let's try to free a bit of memory after the mess we've made with the previous program. Without making any more changes, let's try:

```
BANK 11 CLEAR
```

More likely than not, the system will report **In Use: 0/1**. What has happened? Most likely that the bank itself is in use by the system. Let's try again:

```
BANK 12 CLEAR
```

This time the system will most likely report **0 Ok: 0/1**. We can verify this by running *mem* again. This time it will show us **2 Banks free** (Remember *mem* reports memory in MMT-sized banks, that is 8k). Bank 11 you tried to free originally (unless the system hasn't been modified) is being used by Layer 2 (which takes 3 banks, by default 9, 10 and 11 but can be changed by the **LAYER**, **BANK** command), so it's rightfully marked as *in-use*. Note

here that if you're not using Layer 2, the banks it occupies CAN be used for other purposes including machine code programs. They just cannot be released.

Banks marked as in-use remain reserved after a **NEW** command and are only released at a reset, or with this **BANK n CLEAR**.

BANK CLEAR reports **An invalid Argument** 0:1 if you try to clear banks 1,3,4 and 6 even if you have given the **BANK 1346 JSR** command which is described below.

NextZXS allocates 64K to the RAMdisk by reserving banks 1,3,4 and 6. **BANK 1346 JSR** allows you to release these for use by your programs. Once you give the command

```
BANK 1346 JSR
```

the following things happen: first, all files in the RAMdisk are deleted, then the drive itself is unmounted and using **BANK** commands in these banks stops producing errors. To undo this action and reinstate the RAMdisk you will need to use

```
BANK 1346 FORMAT
```

which will erase the contents of these banks and re-attach them to the RAMdisk. The disk itself however will need to be manually mounted again by using the **MOVE n** command. See Chapter 18 for details.

Bank contents can be copied and erased in whole or in part using the **BANK COPY** and **BANK ERASE** commands. There's also a specific one that copies data quickly to and from the screen but we'll look at that separately. The syntax to copy bank data is

```
BANK source bank COPY [source offset, len] TO destination bank [dest offset]
```

where *source bank* is a read-only bank number to copy from while *destination bank* is a writeable bank number. *Source offset* and *len* signify the location within the source bank and the size in bytes of the memory chunk we're copying. If the latter are specified, then the *dest offset* must also be specified. Let's try

```
BANK 9 COPY TO 47
```

will copy the bank holding the first third of Layer 2 into bank 47 while

```
BANK 1 COPY TO 47
```

will return **An invalid argument** unless **BANK 1346 JSR** has been used.

```
BANK 9 COPY 8192, 8192 TO 47, 0
```

will copy the bottom half of the first third of the Layer 2 screen to the start of bank 47. Once you untangle that tongue-twister you can see how this can create interesting blinds effects.

It's also quite handy to quickly erase the whole or part of a bank, fill it with zeroes or an arbitrary byte value. This is accomplished by the **BANK ERASE** command whose syntax is

```
BANK n ERASE [offset, len], [[value],
```

where *n* is the number of writeable bank, *offset* is the optional starting point of the erase and *len* is the length in bytes of the area to be erased. The optional *value* will fill the area with a byte of your choosing or is omitted (00h). Here are some examples using Layer 2 and an image present in your System/Next™ distribution (you will need to provide the image in 256 x 192 x 256 colour BMP format).

```
10 CD "ENTER HERE THE FOLDER"  
20 LAYER 2,1  
30 ,bnpload yourfile.bmp
```



```

40 BANK 9 COPY TO 111 REM
   first we copy it
50 PAUSE 0 REM wait for a
   key
60 BANK 9 ERASE 128 ; Erase it with
   value 128 which is by default a
   red colour for Layer 2
70 PAUSE 0 , wait for a key
90 BANK 111 COPY TO 9 , restore it
100 PAUSE 0 ; wait for a key
110 LAYER 2 0 LAYER 0

```

You can see easily how as this happens, and how it can be used for a myriad of applications.

Using BANK with graphics

Over the course of chapters dealing with graphics, we've used a lot of graphics related commands that involved the use of BANK. These are BANK LAYER LAYER BANK LAYER PALETTE BANK SPRITE PALETTE BANK SPRITE BANK TILE BANK and TILE DIM all being of great use, significant speed enhancements both in development and in usage.

We saw above the use of BANK COPY to copy data from one bank to another. This includes layer data as they too are kept in banks and managed by AmigaZxOS. There is now even a specially crafted command that does this and more as if it is more optimised specifically tuned to the requirements of display. Unlike BANK COPY this is designed to update small areas of the screen to facilitate effects and display animation. The command is BANK LAYER and is used to quickly copy data from a memory bank to the screen in the current mode or vice versa. The syntax is as follows:

BANK n LAYER x,y,w,h offset TO raster op, offset x,y,w,h

where n is the source OR destination bank number x and y is the upper left character position expressed in character coordinates and row coordinates, w,h are the width and height again in characters of the area to be copied from or copied to, offset is the starting offset in the bank we'll be copying from or to, while raster op is an optional symbol modifier. TO that affects the data being copied at their destination, does not affect the source data.

TO raster op can be one of the following values:

| | |
|------|--|
| TO | Straightforward copy |
| TO & | ANDs the copied data onto the destination |
| TO | ORs the copied data onto the destination |
| TO ^ | XORs the copied data into the destination |
| TO ~ | Copies data into the destination unless it is equal to the global transparency colour (default E3h) if so leaves the destination unchanged |

The area of screen copied by BANK LAYER is defined as with windows in characters. That means the character positions range from 0 to 23 for x and 0 to 23 for y for all modes except LoRes where they range from 0 to 15 for x and 0 to 11 for y.

Data copied from the screen is and on as follows depending upon the currently selected layer (see Chapter 16)

Standard resolution (Layers 0 and 1 1,

The attribute data comes first, stored as h consecutive rows of attributes, w bytes wide. Following this is the screen data, stored as $h \times 8$ consecutive rows of pixel data, w bytes wide. The total memory used is therefore $w \times h \times 9$ bytes.

HiRes (Layer 1,2)

In this mode, each character position is 16 pixels wide, comprising a left and right half. The screen data is stored as $h \times 8$ consecutive pixel rows of data. For each row, the first w bytes comprise the left halves of all characters. The next w bytes in the row comprise the right halves of all the characters. The total memory used is therefore $w \times h \times 16$ bytes.

HiColour (Layer 1 3)

The screen data is stored as $h \times 8$ consecutive pixel rows of data. For each row, the first w bytes comprise the pixel data. The next w bytes in the row comprise the attribute data. The total memory used is therefore $w \times h \times 16$ bytes.

LoRes (Layer 1 0), Layer 2 standard

The data is stored as $h \times 8$ consecutive pixel rows of data. For each row, there are $w \times 8$ bytes, with each byte representing a single pixel. The total memory used is therefore $w \times h \times 64$ bytes.

In the previous section, we dealt with bank management. The following command could very well belong there, but since it deals with memory management of the graphics subsystem and specifically with *layer 2*, we will cover it here. **LAYER BANK** redefines which banks will store Layer 2 display data (the *front buffer*) and which will act as the *back buffer* (for rendering). The syntax is as follows:

LAYER BANK n,m

where n is the front buffer base bank number for Layer 2 (this also sets $n+1$ and $n+2$) and m is the back buffer base bank number, and also sets $m+1$ and $m+2$. These values can be the same and both default to 9. Unlike other **LAYER** commands, it can be executed in any mode. For example to move Layer 2 to banks 13 to 15 (front buffer) and 16 to 18 (back buffer):

```
LAYER BANK 13,16
```

If we now give

```
BANK 9 CLEAR
```

We can see that bank 9 (the original base bank for Layer 2) can now be released. The effects of **LAYER BANK** can be undone either by reversing the command, with **NEW** or with **LAYER CLEAR**.

Memory banks are also ideal to store palette information as palettes are basically a series of 256 bytes or words (depending on your **PALETTE DIM** setting). There are two commands for this: **LAYER PALETTE BANK** and **SPRITE PALETTE BANK**. Their syntax is virtually identical, and is as follows:

LAYER SPRITE PALETTE n BANK $b,offset$

where n is the palette number (0 or 1), b is the bank number and *offset* is the start location in the bank where the palette values are located. As mentioned above, if **PALETTE DIM** was set to 8, **LAYER** and **SPRITE PALETTE BANK** will load 256 bytes from bank b , *offset*, while if **PALETTE DIM** was set to 9, 512 bytes will be loaded.

Apart from the palettes, sprite definitions⁴ themselves can be stored and exchanged through the use of memory banks. The command and its syntax to define either all 64 sprites at once (64 sprites of 256 bytes each equals a full bank of 16K) or some of them is

SPRITE BANK *b [offset, pattern no, number of sprites]*

where *b* is the bank number holding the sprite pattern definitions, *offset* is the starting location in the bank where sprite definitions are stored, *pattern no* is the starting pattern number that's defined by the command and *number of sprites* is the total number of sprites that are defined. If we store all 64 sprite definitions within a bank, then the command can be as simple as

SPRITE BANK 14

which will load 64 sprite definitions from bank 14. Alternatively, to load 32 sprite definitions starting with pattern number 4 from bank 15 offset 256 would require

SPRITE BANK 15,4,256

Sprites and tiles (not to be confused with Layer 3 tiles) are closely related. As a matter of fact as we saw in Chapter 1, their main difference is that tiles are managed by software and not hardware, so it follows that NextBASIC provides similar commands to manage them at least memory-definition wise. The **BANK** commands related to tiles are **TILE BANK** to define the tiles themselves and **TILE DIM** to define the tilemap that is how are the tile patterns organised. The syntax of the first is

TILE BANK *n*

where *n* is the number of the base bank holding the tiles. If more are needed as defined by the tilemap, they will be taken from subsequent bank numbers (up to an additional 3 making a total of 4 banks assigned to tile definitions). The tilemap itself is also held in a bank and managed with

TILE DIM *n,offset w tile size*

which defines the tilemap in bank *n* starting at location *offset* with width *w* which ranges from 1 to 2048 and tile size *tile size* (8 for 8 × 8 pixels or 16 for 16 × 16 pixels).

Using BANK with files

The entire range of **BANK** commands for file management has been covered in length throughout Chapter 21. Next/XLUS and alternatives so we'll just include them here for completeness and as a quick reference. As a general guideline for syntax, **BANK** does not need an *offset* and *length* for **SAVE** operations except the ones that deal with fixed areas. The commands that deal with files and their syntax are

LOAD SAVE VERIFY *filespec BANK n [,offset, length]*

and the additional

SAVE LOAD *filespec LAYER*

that are special shortcut commands to load and save the current layer display. This obviously includes bank address, as for example layer 2 occupies 3 banks and thus it's included here. In all the above, *filespec* is a valid filespec for the filesystem you're accessing, *n* is the bank number while the optional *offset* and *length* must be given together to signify the starting location and length of the data chunk we're manipulating. If omitted, the entirety of the bank is used.

⁴ Although the ZX Spectrum Next's Sprite Engine can define and manipulate a maximum of 256 sprites, these only work with 4-bit palette definitions which are not supported by NextBASIC. Instead NextBASIC supports a total of 64 sprites of 256 colours each.

Extending NextBASIC Programs with BANK

Unlike previous iterations of Sinclair BASIC, NextBASIC makes it possible to write programs larger than the approximate 4 k which is used in the norm with previous ZX Spectrum models. This is achieved through the use of BANK command extensions: whole sections of NextBASIC programs can be copied into any memory bank available to the user and saved/loaded with the **SAVE**, **LOAD**, **BANK** commands as described in Chapter 20 as well as the previous section. Programs can then switch between lines in the "main" program area and those held in a bank.

The following new commands are available to manage banked sections of NextBASIC programs: **BANK LINE**, **BANK LIST** and **BANK LIST PROC()**, **BANK MERGE**, **BANK GO TO**, **BANK GOSUB**, **BANK PROC** and **BANK RESTORE**. We have covered these as well in the appropriate sections of this guide, so they're mentioned here in brief for completeness and reference. Syntax is as follows:

BANK *n* LINE *x,y*

Copies lines *x* through *y* (inclusive) from the main program to bank *n*. The total number of bytes used in the bank will be shown. Once this has been done, it is not possible to change or delete any lines in the banked section, except by completely overwriting the bank's contents using another **BANK LINE** command or by executing a command that will replace the bank's contents with something else.

BANK *n* LIST [*l*] **PROC *name()***

Lists lines (optionally starting with line or label *l*) from a procedure named *name* in bank *n*.

BANK *n* MERGE

Copy all lines back from bank *n* into the main program. This won't overwrite line numbers that did not exist in the source bank.

BANK *n* GO TO

performs a **GO TO** line or label in bank *n*. To **GO TO** a line or label in the main program from a banked section, the bank number should be 255.

BANK *n* GOSUB

branches using **GOSUB** to the subroutine located at line or label in bank *n*. To **GOSUB** to a subroutine in the main program from a banked section, as with **GO TO** above, the bank number should be 255.

BANK *n* PROC *name* (*parameter1*, ..., *parameterN*) TO *variable1*, ..., *variableN*

branches to the **PROC** named *name* located in bank *n* with optional parameters *parameter1*, ..., *parameterN* and optional return values stored in *variable1* to *variableN*. To branch to a **PROC** in the main program from a banked section, as with **GO TO** above, the bank number should be 255.

BANK *n* RESTORE

Sets the **DATA** pointer to line or label in bank *n* ready for the next **READ** operation.

It's noted that **BANK LINE** and **BANK MERGE** can only be given as direct commands and not as part of a saved program held in a bank or in the main section.

NextZXOS Paging Mechanism Overview

As we discussed in the introduction to this chapter, the CPUs used in all previous models of the ZX Spectrum line as well as this one, can only address 65536 bytes. The original 28x ZX Spectrum crammed in more than twice the amount of memory than it could address, clocking in at 131072 bytes of RAM and 32768 bytes of ROM, making 163840 bytes

49K 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

All of the memory is divided into 16K banks. The first 16K bank is reserved for the OS. The next 16K bank is reserved for the user. The next 16K bank is reserved for the user. The next 16K bank is reserved for the user.

NextBASIC and NextZXOS, and it's reserved for CP/M)

The banks are divided into 16K banks. The first 16K bank is reserved for the OS. The next 16K bank is reserved for the user. The next 16K bank is reserved for the user. The next 16K bank is reserved for the user.

choose which part is relevant

RAM is really rather useful

NextZXOS is a 16K bank. The first 16K bank is reserved for the OS. The next 16K bank is reserved for the user. The next 16K bank is reserved for the user. The next 16K bank is reserved for the user.

understand the underlying mechanisms we can elaborate a little bit

The banks are divided into 16K banks. The first 16K bank is reserved for the OS. The next 16K bank is reserved for the user. The next 16K bank is reserved for the user. The next 16K bank is reserved for the user.

(meaning the CPU has their exclusive use)

The banks are divided into 16K banks. The first 16K bank is reserved for the OS. The next 16K bank is reserved for the user. The next 16K bank is reserved for the user. The next 16K bank is reserved for the user.

Assuming the user has a bank, the user can use it for their own purposes.

REG 8, % REG 8 001000000

The user can use the bank for their own purposes. The user can use the bank for their own purposes. The user can use the bank for their own purposes.

The user can use the bank for their own purposes. The user can use the bank for their own purposes. The user can use the bank for their own purposes.

The user can use the bank for their own purposes. The user can use the bank for their own purposes. The user can use the bank for their own purposes.

The user can use the bank for their own purposes. The user can use the bank for their own purposes. The user can use the bank for their own purposes.

machine only the top slot (slot 4) of the address space was banked in and only the user located at address range C000h to FFFFh.

When the ZX Spectrum +3 came out, there were two more 16K ROMs introduced, which didn't originally exist, that paired with the need to run CP/M that requires RAM at the bottom of the address map necessitated the creation of yet another IO address: 7FFDh.

Between these two ports, there are enough bits to address all the RAM pages of an unexpanded Next; however, on a fully expanded Next, one more port was needed to be able to address the entire physical memory available. These methods are all extending one another so backwards compatibility is ensured, while the introduction of the MMLs allows for a more straightforward memory management system for user programs.

Let's begin now, this all works by first looking at 28K style paging. The hardware port that

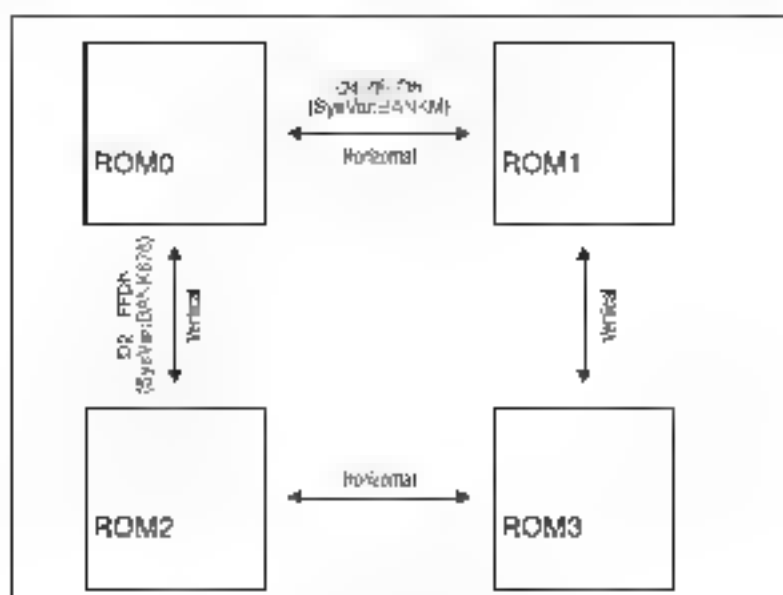


Fig. 40 Horizontal vs Vertical ROM switching

controls is at IO address 7FFDh (32765). The bit layout for this port is as follows:

| Bit | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|-------------|----|----|-------------------|---------------|------------------|------------|----|----|
| Description | | | Disable
Paging | ROM
Select | Screen
Select | RAM Select | | |

D2 to D0 is a three bit number that selects which RAM page goes into the C000h to FFFFh slot in previous models, such as the 7 for in BASIC. RAM page 0 was normally in slot and when editing, RAM page 7 was paged in for various buffers and scratchpads.

D3 switches screens. Screen 0 (the Display + Colour Files) was held in RAM5 (beginning at 4000h) and it was the one that BASIC used; screen 1 was held in RAM7 (beginning at C000h) and could only be used by machine code programs.

D4 determines whether ROM0 (the editor ROM) or ROM1 (the 48k BASIC ROM) is paged into Slot 1 at 0000h to 3FFFh.

D5 is a safety feature: once this bit is set, no further paging operations will work. This is normally used when the machine assumes a standard 48K Spectrum configuration and all the memory paging circuitry is locked out. On previous models, this meant that it couldn't be turned back into a 28K machine other than by rebooting, however, the sound chip can still be driven by OUT either from 48k Basic or machine code. On the ZX Spectrum Next, however, you can override that lock switch it back to on by setting NextREG 8, D7 to 1.

Note here that the 16K Bank 5 is the bank read by the JLA to determine what to show on screen for Layer 0 and . The JLA connects directly to the larger memory space ignoring mapping; the screen is always 16K Bank 5, no matter where in memory it is, or if it is switched in at all! Setting D3 of Memory Paging Control (7FFDh) will have the JLA read instead from 16K Bank 7 (otherwise known as 'shadow screen' which can be used as an alternate screen. Beware that this does not map 16K bank 7 into RAM; to alter 16K bank 7 it must be mapped by other means.

Let's now examine the bit layout of port 1FFDh used by the .

| B0 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|-------------|----|----|----|----------------------|-----------------|-------------|---------------------|----|
| Description | | | | Par. Port
Status? | Disk
Master? | Switch type | ROM / RAM switching | |

When D0 is 0, D1 has no effect and D2 is a 'vertical' ROM switch (ie between ROM0 and ROM2 or between ROM1 and ROM3). D4 at 7FFDh on the other hand is a 'horizontal' ROM switch (ie between ROM0 and ROM1 or between ROM2 and ROM3). The following diagram illustrates the various ROM switching possibilities.

It is best to think of D4 in port 7FFDh and D2 in port 1FFDh combining to form a 2-bit number ranging from 0 to 3, which determines which ROM occupies the memory area 0000h to 3FFFh. 16K Slot 1. D4 of port 7FFDh is the least significant bit and D2 of 1FFDh is the most significant bit.

| D2/1FFDh | D4/7FFDh | ROM Used |
|----------|----------|----------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 2 |
| 1 | 1 | 3 |

ROM switching (with D0 of 1FFDh set to 0)

Tying it all together, we can easily surmise that 128 style memory management can only alter the bank address at 0000h for 16K banks (that would be Slot 4, or for 8K MMIO type banks Slots 7 and 8). The active 16K bank at 0000h is selected by writing the 3 LSBs of the 16K bank number to the bottom 3 bits of Memory Paging Control (7FFDh) and the 4 MSBs to the bottom 4 bits of Next Memory Bank Select (DFFDh). The reason for the division is that the original Spectrum 128, having only 128k of memory, only needed 3 bits.

This in essence constructs a 'super hardware port' of sorts, very similar to the combination used to select a ROM using bits from 1FFDh and 7FFDh.

| D3/DFFDh | D2/DFFDh | D1/DFFDh | D0/DFFDh | D2/7FFDh | D1/7FFDh | D0/7FFDh | Bank |
|----------|----------|----------|----------|----------|----------|----------|------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 2 |
| 1 | 1 | 1 | 1 | 1 | 1 | | 127 |

"Standard Next paging" bank selection settings

If you are using the standard interrupt handler or NextZXOS routines, then any time you write to the Memory Paging Control port (7FFDh), you should also store the value in Sysvars at location 5B5Ch. Any time you write to the +1 Memory Paging Control (1FFDh), you should also store the value at 5B67h. There is no corresponding system variable for the Next-only Next Memory Bank Select (DFFDh) port.

5 Not applicable on the ZX Spectrum Next.

Note that internally NextXIOS and NextRASIC will select a combination of all possible banking methods according to what's needed at which time, and you should not rely on this information as a definitive guide on how the system behaves at all times.

allRAM mode

A special paging mode, also called **allRAM mode** or **CP/M mode**, is enabled by writing a value with bit 5 set in the 9 Memory Paging Control (1FFDh). Depending on the 9 low bits of this value a memory configuration is selected as follows:

| D2: 1FFDh | D1: 1FFDh | D0: 1FFDh | RAM Page combinations (Slot0...Slot4) |
|-----------|-----------|-----------|---------------------------------------|
| 0 | 0 | | 0 2 3 |
| 0 | 1 | 1 | 4 5 6 7 |
| | 0 | | 4 5 6 3 |
| | | | 4 7 6 3 |

allRAM paging

This mode is selected by default when you select the *CP/M Menu* from the *More* submenu of the *Startup menu*, or you run the dot command `cpm`.

MMU-Based Memory Management

MMU-Based memory management is much simpler to use. It only requires a write in the appropriate MMU Next Register to change the 8k bank occupying a specific 8k slot in the 64K address space. See the previous chapter for details on Next Registers. The MMU registers begin with slot 0 in NextREG 80 (50h) and end with slot 7 in NextREG 87 (57h). For MMU0 and MMU1 only, the ROM can be paged in by selecting 255 (FFh) as a bank number. The default values for the MMU Registers are listed in Chapter 29 and correspond to the normal default memory mapping of the 128K Spectrum.

Layer 2 Bank Switching

Layer 2 can also be overlaid on top of the MMU memory map in the bottom 16K or 48K in a Read-only or Write-only mapping. The Write-only mapping, for example, would mean that memory writes in the bottom 16k go in Layer 2 but memory reads come from the MMU mapping as normal. The bottom 16k is normally occupied by the ROM so this Write-only mapping would allow NextRASIC programs to continue to run in the ROM as a read-only program while allowing POKEs to write into the Layer 2 screen. This is an easy way to gain access to 32k in a single 16K address range.

The Layer 2 mapping is controlled by bits in the *Layer 2 Access Port 4667* (120Bh). These bits select among 16K or 48K mapping, Read-only or Write-only, and whether the active Layer 2 screen is mapped to a second Layer 2 buffer. Once new screen is mapped, Layer 2 and its second buffer can be located anywhere in RAM and the starting 16K banks are programmed into NextREG 18 (12h) and 19 (13h) respectively.

The Layer 2 mapping does not have to be used for Layer 2 graphics only. It can be used as a third banking mechanism to access memory more generally.

Paging method interactions

The most recent change to the memory map, whether that is by Standard or MMU methods, always applies. Each time a change is made to the memory map using the Standard mechanism, a write to port 7FFDh (DFFDh) or 1FFDh, the affected MMU is changed immediately. For example, writing to port 7FFDh will change MMU0 and MMU1 to FFh to make sure the selected ROM is visible and MMU6 and MMU7 will be changed to reflect the selected 16k RAM bank.

Paging out the ROM

As seen above, the ROM can be paged out by enabling **allRAM** mode, or by using MMU-based memory management. This may cause problems as some programs may assume

that ROM-based service routines are present at fixed addresses in ROM. Additionally, if the default interrupt mode `IM` is set, the CPU will JP to `0038h` every frame trying to find an interrupt handler routine. If it does not (which it won't unless you write your own), the system will crash.

Chapter 24 The System Variables

Overview

Certain locations in memory are set aside for specific uses by the system. There are a few routines used to keep the paging in order, and some locations called system variables (or SYSVARS). You can use PEEK and DPEEK to read them in order to find out various things about the system, and on some of them you can usefully change with POKE and DPOKE. They are listed here with their uses.

The area occupied by SYSVARS spans the addresses 23296 (5B00h) to 23733 (5CB5h), inclusive, and a subset of them are used in 48K BASIC: addresses 23552 (5C00h) to 23733 (5CB5h).

Note that in 48k mode, there is a buffer area between 23296 (5B00h) and 23552 (5C00h) which was used for controlling the printer. This was quite a popular location for small machine code programs on the old 48K Spectrum, and if any of these routines are tried in NextBASIC, the computer will invariably crash. It's advisable that any 48K BASIC program that uses PEEK, POKE and JSR to either be run in 48k BASIC mode, although it can be entered in NextBASIC mode and transferred using the SPECTRUM command, or examined thoroughly and converted so it won't use any of these commands. Alternatively, any machine code routine embedded within it should be moved to a safer area.

System Variables

The system variables listed below all have unique names, but do not confuse them with NextBASIC variables. The computer will not recognize the names as referring to system variables, and they are given solely as mnemonics to be human-readable.

The abbreviations in column 1 of the table that follows have the following meanings:

- X** The variables should not be poked because the system might crash
- N** Poking the variable will have no lasting effect
- R** Routine entry point. Not a variable

The number in column 1 is the number of bytes in the variable. For two bytes, the first one is the least significant byte, the reverse of what you might expect, so to POKE a value *v* to a two byte variable at address *n*, use DPOKE instead, as it does the conversion for you. Otherwise you'll need to enter the following for SYSVAR *n* value *v*:

```
POKE n,v-255*INT (v/256)
POKE n+1,INT (v/256)
```

and to peek its value, either use DPEEK or the expression

```
PEEK n+255*PEEK (n+1)
```

| Notes | Address | | Name | Description |
|-------|---------|-------|-----------|--|
| | Hex | Dec | | |
| R16 | 5B00 | 23296 | SWAP | Paging subroutine |
| R17 | 5B00 | 23296 | STOO | Paging subroutine. Entered with interrupts already disabled and AF BC on the stack |
| R9 | 5B20 | 23328 | YOUNGER | Paging subroutine |
| R16 | 5B2A | 23328 | REGNOQY | Paging subroutine |
| R24 | 5B3A | 23354 | ONEAR | Paging subroutine |
| X2 | 5B52 | 23378 | OLDHL | Temporary register store while switching ROMs |
| X2 | 5B54 | 23380 | OLDBC | Temporary register store while switching ROMs |
| X2 | 5B58 | 23382 | OLDAF | Temporary register store while switching ROMs |
| X | 5B58 | 23384 | CACHEBANK | 8K Bank ID holding address register data |
| N1 | 5B58 | 23386 | | Reserved for system use |

| Notes | Address | | Name | Description |
|-------|---------|-------|-----------|--|
| | Hex | Dec | | |
| X2 | 5B5A | 23386 | RETADDR | Return address in ROM |
| X1 | 5B5C | 23388 | BANKM | Copy of last byte output to I/O port 7FFDh (32765). See Chapter 23 The Memory. This byte must be kept up to date with the last value output to the port if interrupts are enabled. |
| X* | 5B5D | 23389 | RAMRST | RST 6 instruction. Used by ROM to report errors in ROM 3. |
| N1 | 5B5E | 23390 | RAMERR | Error number passed from ROM 1 to ROM 3. Also used by SAVE/LOAD as temporary drive store. |
| 1 | 5B5F | 23391 | INK1 | INK colour for LoRes. |
| 1 | 5B60 | 23392 | INK2 | INK colour for Layer 2. |
| 1 | 5B61 | 23393 | ATTRULA | Attributes for standard mode. |
| 1 | 5B62 | 23394 | ATTRHR | Attributes for HiRes (only paper colour in bits 5:3 is used). |
| 1 | 5B63 | 23395 | ATTRHC | Attributes for HiColour. |
| 1 | 5B64 | 23396 | INKMASK | Softcopy of EnhancedULA Inkmask (or 0). |
| N* | 5B65 | 23397 | LSBANK | Temporary bank number in LOAD/SAVE and other operations. |
| 1 | 5B66 | 23398 | FLAGS3 | Various flags. Bits 0-4 and 7 unlikely to be useful. Bit 2 is set when tokens are to be expanded or printing. Bit 3 is set if print output is RS232. The default (at reset) is Centronics. Bit 4 is set if a disk interface is present. Bit 5 is set if drive B is present. |
| X1 | 5B67 | 23399 | BANK87B | Copy of last byte output to I/O port 1FFDh (8-89). This port is used to control the 3-extended RAM and ROM switching (bits 0-2). If bit 0 is 0 then bit 2 controls the vertical ROM switch (2 and 3) the disk motor (bit 3) and Centronics strobe (bit 4). This byte must be kept up to date with the last value output to the port if interrupts are enabled. |
| X | 5B68 | 23400 | FLAGN | Flags for the NextZXOS system. |
| 1 | 5B69 | 23401 | MAXBANK | Maximum available RAM bank. |
| X2 | 5B6A | 23402 | OLDSP | Old SP (stack pointer) when STACK is in use. |
| X2 | 5B6C | 23404 | SYNRET | Return address for ONERR. |
| 5 | 5B6E | 23406 | LASTV | Last value printed by calculator. |
| 1 | 5B73 | 23411 | TILEBANK1 | Tiles bank for LoRes. |
| 1 | 5B74 | 23412 | TILEM1 | Tilemap bank for LoRes. |
| 1 | 5B75 | 23413 | TILEBANK2 | Tiles bank for Layer 2. |
| 1 | 5B76 | 23414 | TILEM2 | Tilemap bank for Layer 2. |
| X* | 5B77 | 23415 | NOTBANK | Bank containing NXTLIN. |
| X | 5B78 | 23416 | DATABANK | Bank containing DATAD0. |
| 1 | 5B79 | 23417 | LOADRV | Holds 'T' if LOAD/VERIFY/MERGE are from tape, otherwise holds 'A', 'B' or 'M'. |
| 1 | 5B7A | 23418 | SAVDRV | Holds 'T' if SAVE is to tape, otherwise holds 'A', 'B' or 'M'. |
| N* | 5B7B | 23419 | L2SOFT | Softcopy of Layer 2 font. |
| 2 | 5B7C | 23420 | TILEW1 | Width of LoRes tilemap. |
| 2 | 5B7E | 23422 | TILEW2 | Width of Layer 2 tilemap. |
| 2 | 5B80 | 23424 | TILEOFF1 | Offset in bank for LoRes tilemap. |
| 2 | 5B82 | 23426 | TILEOFF2 | Offset in bank for Layer 2 tilemap. |
| 2 | 5B84 | 23428 | COORDSX | X Coordinate of last point plotted (Layer 1/2). |
| 2 | 5B86 | 23430 | COORDSY | Y Coordinate of last point plotted (Layer 1/2). |
| 1 | 5B88 | 23432 | PAPER1 | PAPER colour for LoRes mode. |
| 1 | 5B89 | 23433 | PAPER2 | PAPER colour for Layer 2 mode. |
| N* | 5B8A | 23434 | TEMPVARS | Base of temporary system variables (space shared with bottom of STACK). |

| Notes | Address | | Name | Description |
|-------|---------|-------|---------|--|
| | Hex | Dec | | |
| X117 | 5BFF | 2355 | TSTACK | Temporary stack grows down from here. Used when RAM bank 7 is switched in at top of memory while executing the editor or calling NextZKUS. It may safely go down to 5BBAh if necessary. This guarantees at least 7 bytes of stack when NextBASIC calls NextZKUS. |
| N8 | 5C00 | 2356 | KSTATE | Used in reading the keyboard. |
| N1 | 5C08 | 23560 | LASTK | Stores newly pressed key. |
| 1 | 5C09 | 23561 | REPDEL | Time (in 50 th of a second) that a key must be held down before it repeats. This starts off at 35, but you can POKE in other values. |
| | 5C0A | 23562 | REPPER | Delay (in 50 th of a second) between successive repeats of a key held down. Initially 5. |
| X2 | 5C0B | 23563 | RETVAR5 | Address of local variables on return stack. |
| N1 | 5C0D | 23565 | K DATA | Stores 2 nd byte of colour controls entered from keyboard. |
| N2 | 5C0E | 23566 | TVDATA | Stores bytes of colour, AT and TAB controls going to TV. |
| X28 | 5C10 | 23568 | STRMS | Addresses of channels attached to streams. |
| 2 | 5C36 | 23606 | CHARS | 256 less than address of character set (which starts with space and carries on until 0). Normally in ROM, but you can set up your own in RAM and make CHARS point to it. |
| | 5C38 | 23608 | RASP | Length of warning buzz. |
| | 5C39 | 23609 | PIP | Length of keyboard click. |
| | 5C3A | 23610 | ERRNR | 1 less than the report code. Starts off at 255 (for 1, so PEEK 25610 gives 255). |
| X1 | 5C3B | 23611 | FLAGS | Various flags to control the NextBASIC system. |
| X1 | 5C3C | 23612 | TVFLAG | Flags associated with the TV. |
| X2 | 5C3D | 23613 | ERRSP | Address of item on machine stack to be used as error return. |
| N2 | 5C3F | 23615 | | Reserved for system use. |
| N1 | 5C41 | 23617 | MODE | Specifies K, L, D, E or G cursor. |
| 2 | 5C42 | 23618 | NEWPPC | Line to be jumped to. |
| N1 | 5C44 | 23620 | | Reserved for system use. |
| 2 | 5C45 | 23621 | PPC | Line number of statement currently being executed. |
| 1 | 5C46 | 23622 | SUBPPC | Number within line of statement currently being executed. |
| | 5C48 | 23624 | BORDCR | Border colour multiplied by 8; also contains the attributes normally used for the lower half of the screen. |
| 2 | 5C49 | 23625 | E PPC | Number of current line with program cursor. |
| X2 | 5C4B | 23627 | VARS | Address of variables. |
| N2 | 5C4D | 23629 | DEST | Address of variable in assignment. |
| X2 | 5C4F | 23631 | CHANS | Address of channel data. |
| X2 | 5C51 | 23633 | CURCHL | Address of information currently being used for input and output. |
| X2 | 5C53 | 23635 | PROG | Address of NextBASIC program. |
| X2 | 5C54 | 23637 | NEXTLIN | Address of next line in program. |
| X2 | 5C57 | 23639 | DATADD | Address of terminator of last DATA item. |
| X2 | 5C59 | 23641 | E_LINE | Address of command being typed in. |
| 2 | 5C5B | 23643 | K_CUR | Address of cursor. |
| X2 | 5C5D | 23645 | CH_ADD | Address of the next character to be interpreted. (This character alters the argument of PEEK, or the NEWLINE at the end of a POKE statement). |
| 2 | 5C5F | 23647 | X_PTR | Address of the character after the 0 marker. |
| X2 | 5C61 | 23649 | WORKSP | Address of temporary work space. |
| X2 | 5C63 | 23651 | STKBOT | Address of bottom of calculator stack. |
| X2 | 5C65 | 23653 | STKEND | Address of start of spare space. |
| N1 | 5C67 | 23655 | BREG | Calculator's B register. |

| Notes | Address | | Name | Description |
|-------|---------|-------|-----------|--|
| | Hex | Dec | | |
| N2 | 5C68 | 23656 | MEM | Address of area used for calculator's memory (usually MEMBOT, but not always) |
| | 5C6A | 23658 | FLAGS2 | More flags: Bit 3 set when CAPS SHIFT or CAPS LOCK is on |
| X | 5C6B | 23659 | DF SZ | The number of lines (including one blank line) in the lower part of the screen |
| N2 | 5C6C | 23660 | | Reserved for system use |
| 2 | 5C6E | 23662 | OLDPPC | Line number to which CONTINUE jumps |
| | 5C70 | 23664 | OSPPC | Number within line of statement to which CONTINUE jumps |
| N | 5C71 | 23665 | FLAGX | Various flags |
| N2 | 5C72 | 23666 | STRLEN | Length of string type destination in assignment |
| N2 | 5C74 | 23668 | T ADDR | Address of next item in syntax table (very unlikely to be useful) |
| 2 | 5C75 | 23670 | SEED | The seed for RAND: This is the variable that is set by RANDOMIZE |
| 3 | 5C78 | 23672 | FRAMES | 3 byte (least significant byte first), frame counter incremented every 20ms |
| 2 | 5C79 | 23673 | UDG | Address of first user-defined graphic. You can change this (for instance) to save space by having fewer user-defined characters |
| 1 | 5C7D | 23677 | COORDS | X-coordinate of last point plotted |
| 1 | 5C7E | 23678 | | Y-coordinate of last point plotted |
| X1 | 5C7F | 23679 | GMODE | Graphical layer/mode flags |
| X | 5C80 | 23680 | PRCC | Full address of next position for LPRINT to print at (in ZX Printer buffer). Legal values 58C0-5B1F |
| | 5C81 | 23681 | \$TIMEOUT | Screensaver control |
| 2 | 5C82 | 23682 | ECHO E | 33-column number and 24 line number (in lower half) of end of input buffer |
| 2 | 5C84 | 23684 | DF GC | Address in display file of PRINT position |
| 2 | 5C86 | 23686 | DF COL | Like DF GC for lower part of screen |
| X | 5C88 | 23688 | S POSN | 33-column number for PRINT position |
| X | 5C89 | 23689 | | 24-line number for PRINT position |
| N2 | 5C8A | 23690 | SPOSNL | Like S POSN for lower part |
| | 5C9C | 23692 | SCR CT | Counts scrolls: it is always more than the number of scrolls that will be done before stopping with scroll? If you keep doing this with a number bigger than (say 255) the screen will scroll on and on without asking you |
| | 5C8D | 23693 | ATTR P | Permanent current colours, etc. (as set up by colour statements) |
| | 5C8E | 23694 | MASK P | Used for transparent colours, etc. Any bit that is 1 shows that the corresponding attribute bit is taken not from ATTR P, but from what is already on the screen |
| N1 | 5C8F | 23695 | ATTR T | Temporary current colours, etc. (as set up by colour items) |
| N | 5C90 | 23696 | MASK T | Like MASK P, but temporary |
| | 5C91 | 23697 | P FLAG | More flags |
| N30 | 5C92 | 23698 | MEMBOT | Calculator's memory area (used to store numbers that cannot conveniently be put on the calculator stack) |
| 2 | 5C80 | 23728 | | Not used |
| 2 | 5C82 | 23710 | RAMTOP | Address of last byte of NextBASIC system area |
| 2 | 5C84 | 23712 | P RAMT | Address of last byte of physical RAM |

Not used in Z80 mode or when no external peripherals are attached

Chapter 25 Using Machine Code

Using Machine Code

[illegible]

But the code will perform a value of 0 and the code might need to be modified to suit the specific computer system as well as assembly language with which you are working. It is important to note that the code is not understood by the ZX Spectrum Next's CPU in Appendix A.

[illegible]

Let's take as an example the program

99

Now we'll do the BC register, but with 99 and the other 11 instructions into the 16-bit code. The bytes 99 00 bc 99 and 201 ref. If you look at codes 1 and 20 in 4 bits, the 4 you will find that 00 is 00 and bc NN where NN is just 00 or any other number, and 201 corresponds to ref.

Using CLEAR to Make Space

The pay was withheld until the employee signed the pay check in hand. The employee
 paid a sum of money to the employer, which was a sum of money, and if the employee is
 to take a sum of money, the employee is to take a sum of money, and if the employee is

If you enter the command

CLEAR 05207

This will give you a space of 100 ft² good measure. Be sure and add these 65268 to the 65268 that you already have. The total is 130536. This is the total area of the field.

```

10  A=65265
20  READ  R   POKE  A,R

```

* A cross-assembly is an assembly that runs on a different system than the one it originally came for. For example, a 286 ISA card may run on a 386 or 486 system.


```
30  a+=1  GO TO 20
40  DATA 1,99,0,201
```

This will stop with the report **E Out of DATA** when it has filled in the four bytes you specified.

Using JSR to run machine code

To run the machine code, you use the function **JSR** or its preferred, **BANK** command variant. In its simplest form **JSR** must be provided with a numeric argument, i.e. the starting address or the bank offset. Its result is the value of the **BC** register on return from the machine code program, so assuming you type

```
PRINT USR 65266
```

will return the value 99.

The return address to *NextBASIC* is stacked in the usual way, so returns by a **Z80 ret** instruction. You should not use the **Y** and **I** registers in a machine code routine that expects to use the *NextBASIC* interrupt mechanism. To perform the exact same function by using the **BANK** variant, make the following changes to our program:

```
10  %a=0
20  BANK NEW %b
30  READ %n      BANK %b POKE %a %n
40  %a+=1  GO TO 30
50  DATA 1,99,0,201
```

RUN it and you'll see the **E Out of Data** error again. Now it's time to execute it and it's done by giving

```
PRINT % BANK b JSR 0
```

There are a few more variants of **USR** that differ in key points and make the life of the machine-code programmer a bit easier. These are

```
USR$(addr)
BANK n USR$ offset
```

which call the machine code routine at *addr* (or *offset* in bank *n*). Instead however of returning the 16-bit number found in **BC** (as with **USR addr**), **USR\$** returns a string, defined by the start address returned by the machine code routine in **DE** and length in **BC**.

Additionally, **USR** as well as **USR\$** and their **BANK** variants, can be provided with optional parameters like

```
USR(addr param1 [ param2 [ param3 ] ],
USR$(addr param1 param2 [ param3 ])
BANK n USR(addr param1 param2 [ param3 ])
BANK n USR$(addr param1 param2 [ param3 ] )
```

which can be passed to the machine-code routine instead of just the start address in **BC**.

If a single additional parameter (*param1*) is present, this is passed in **BC** (if it is numeric) or as an address **DE** and length **BC** (if it is a string).

The type of the parameter passed to the routine is indicated by the zero flag: if set, the parameter is a string (in **DE-BC**), otherwise the parameter is a number in **BC**.

[illegible]

ADD HL,HL Type bits are 0 for string, 1 for numeric

be unbalanced and the expression may be calculated incorrectly

[illegible]

variables BANKM and BANK678 for 7FFDh and 1FFDh respectively

6384 4000m

Yn₁ can save v₁ 78 time node p1 2 3m 435 h, p1. right with example

SALE "name" CODE 65266.4

or in case you used the BANK variant

SEE "Name" BOX 2b, p. 4

There is no way of knowing the exact day a ship will arrive and it is unfortunately not so easy to know when the ship will leave. It is not possible to know when the ship will leave.

```
10 LOAD 'name' CODE 65268,4
20 PRINT USR 65268
```

Any test should also be conducted at separate intervals to ensure that the test is valid.

SAVE "loader" LINE 10

Y. H. Ma, R. Ma, H. Yang, J. Wang, Y. Bai, B. Li, C. Li, J. Ma, S. Li, Y. Wang

LOAD loader

It's more just to avoid it all together. Let your BANK tell you what it can best do and runs the money for you. It's an easy and more advantageous way to let the BANK do what it does best and that's safer and always preferred.

Calling NextZXOS from NextBASIC

When **NextBASIC's** **USR** function is used, the code it references is entered with the memory configured with the ROM switched in at the bottom of memory in the address range 0000h - 3FFFh, being ROM 3 (the 48 BASIC ROM). The RAM page at the top of memory is Bank 0 and the machine stack resides in this area - unless the **CLEAR** command has been used to reduce it to somewhere below C000h. As explained in the accompanying documents explaining the **NextZXOS** API (found in the c:\docs\nextzxos\ folder in your System/Next "distribution"), **NextZXOS** can only be called with RAM page 7 switched in at the top of memory, the stack held somewhere in that range 4000h to BFE0h, and ROM 2 (the **NextZXOS** ROM) switched in at the bottom of memory (0000h to 3FFFFh).

Consequently, it will be necessary to switch both ROM and RAM, and move the stack before and after calling one of the entries in the DOS jump table.

† the **CLEAR** command has been used so that the NextBASIC stack is below 49120 (\$BFE0h); then it is not necessary to move the stack. However, we have done so in the following example to demonstrate the technique when this is not the case.

A simple example to call DOS CATALOG

[illegible]


```

ld bc 1044
ld hl 0
ldir
ld b 64
ld c 1
ld de ca 1111
ld hl 5
startdat
call dos ca 1111
push a

pop a
d doreset ah

d c b
d b

a.
push b
ld a
a, ca 1111
set t a
and 111h
ld hl 5
ld sp startdat
ret

startdat
db 0 "A" 55h
    "L" 11h

dreset
defw 0

```

As some of you may not have an assembler available, the following is a NextBASIC program that pokes the above code into memory, calls it, and then uses the value returned by the **JSR** function and the contents of **dosret** to print a very simple catalog of the disk.


```

10 SUM=0
20 FOR I=28672 TO 28758
30   READ N
40   POKE I,N   SUM+=N
50 NEXT I
60 IF SUM <> 9367 THEN PRINT
   'Error in DATA'   STOP
70 X= LSR 28672
80 IF INT ( PEEK (28757)/2)=
   PEEK (28757)/2 THEN PRINT
   'Disk Error ',PEEK
   (28758),   STOP
90 IF X=1 THEN PRINT 'No file
   found'   STOP
100 FOR I=0 TO X-2
110   FOR J=0 TO 10
120     PRINT CHR$( I PEEK
       (32761+i*13+j)),
130   NEXT J
140   PRINT
150   NEXT I
160 DATA 243,237,115 0,144,1,
       253,127,58,92,91 203,167,2
       45,7,50 92,91,237 121,49,2
       55,159 251
170 DATA 33 0,128,17,1,128,1,
       0,4,54 0,237,178 6,64,14,1
       ,17,0 128,33,61,112,205,30
       ,1,245 225,34,85 112,72,0
       ■
180 DATA 243,197,1,253,127,58,
       92,91 203 231,230 248,50,9
       2,91,237,121,193 237,123,0
       , 144,201
190 DATA 42 45,42,255 0,0

```

The addresses picked for the above code and its data areas are completely arbitrary. However, it is a good idea to keep things in the central 32K wherever possible so as not to run into the pitfall of accidentally switching out a vital variable or place of code.

If interrupts are to be enabled, (as is the case in the above example), it is imperative that the system is kept up to date about the latest ROM switch. This means that the user must make the BANK678 system variable reflect the last value output to the port at 1FFDh. As shown by the above example, the general technique is to take a copy of the variable in A, self-reset the relevant bits, update the system variable, then make the switch with an OUT instruction. Interrupts must be disabled while the system variable does not reflect the cur-

rent state of the port. The port at 1FFDh doesn't just control the ROM switch, so setting the variable to absolute values would be very unwise. Using AND/OR with a bit mask or SET/RES instructions is the preferred method of updating the variable.

Just as BANK678 reflects the last value output to 1FFDh, BANKM should also be kept up to date with the last value output to 7FFDh. Again, it is unwise to use absolute values, as the port is used for other purposes. For example, the bottom 9 bits of the port are used to select the RAM page that is switched into the memory area C000h through FFFFh (this is also shown in the above example). Naturally, when more than one bit is to be set/reset, a bit mask used with OR/AND is the more efficient method. Note that RAM paging was described in the *Memory Management* section in Chapter 24.

The above was a very simple example of calling DOS routines. It works apart from the ZX Spectrum Next on the ZX Spectrum +3 and ZX Spectrum +3e as well.

Opcodes Prefixes

Some Assembler opcodes are preceded by a prefix byte which changes the opcode represented by the following byte.

Assembler opcode prefixes CBh (203) and EDh (237) alter the meaning of certain instructions, as indicated in the 5th and 6th columns of Appendix A. This includes the provision of some entirely new opcodes for the ZX Spectrum Next.

Assembler opcode prefixes DDh (221) and FDh (253) alter the meaning of certain instructions that ordinarily refer to the H or L registers, so that they refer to either the component registers or IX or IY register respectively. For example, the instruction LD H,n will load the value of n into the H register. Preceding this two-byte instruction with the IX register's opcode prefix DDh would result in the most significant 8 bits of the X register being loaded with that value instead.

This general transformation rule is modified when the original instruction contains HL with this component replaced by (IX + N) and any other reference to HL left unaffected. For instance:

DDh 66h is interpreted as **ld h,(ix + N)**

A DDh opcode will be ignored (interpreted as nop) if it precedes DDh, EDh or FDh. Similar rules apply to the FDh instruction.

Appendix A

Character Set, Z80N Mnemonics and Control Codes

This is the complete ZX Spectrum Next Next/XOS character set with codes in decimal and hexadecimal, the character each code represents, as well as the control codes shaded together with their corresponding NextBASIC tokens, if any. Tokens that are shaded are specific to the ZX Spectrum Next and cannot be found in earlier ZX Spectrum models. As the codes are also Z80N machine code instructions, the right hand columns give the corresponding assembly language mnemonics. As you are probably aware if you understand these things, certain Z80N instructions are compounds starting with **CB** or **ED**, the two rightmost columns give you these. Note that **ED** instructions that are shaded cannot be found in regular Z80 CPUs and are only native to Z80N, the variant of the Z80 CPU found on the ZX Spectrum Next. Control codes are marked with **JW** if they refer to User Windows and **SW** if they refer to System Windows.

| Dec | Character | Control Code | Token | Hex | Z80N Assembler | after CB | after ED |
|-----|---|--------------|-------|-----|----------------|----------|----------|
| 0 | Justify off (JW) Increase font (SW) | | | 00 | hlp | rlc b | |
| 1 | Justify on (JW) Decrease font (SW) | | | 01 | kl bc,NN | rlc c | |
| 2 | Save Window (JW) Change font (SW) | | | 02 | kl (bc),a | rlc d | |
| 3 | Restore Window (JW)
Regenerate Small Fonts (SW) | | | 03 | rlc bc | rlc e | |
| 4 | Cursor to top left (JW)(SW) | | | 04 | rr b | rlc h | |
| 5 | Cursor to bottom left (JW)(SW) | | | 05 | dec b | rl | |
| 6 | PRINT comma | | | 06 | kl b,N | rlc (hl) | |
| 7 | EDIT Scroll (SW)(JW) | | | 07 | rlca | rlc a | |
| 8 | => | | | 08 | ex hl,hl' | rrc a | |
| 9 | = | | | 09 | add hl,bc | rrc c | |
| 0 | 0 | | | 0A | kl a (hl) | rrc d | |
| | 9 | | | 0B | dec b | rrc e | |
| 2 | DELETE Backspace | | | 0C | rr | rrc h | |
| 3 | ENTER Carriage Return
PRINT apostrophe | | | 0D | dec | rrc | |
| 4 | Clear Window (JW)(SW)
Wash Window (JW)(SW) | | | 0E | kl a,N | rrc (hl) | |
| | | | | 0F | rrca | rrc a | |
| 6 | OK | | | 10 | djnz DIS | rl b | |
| | PAPER | | | 1 | kl bc,NN | rl c | |
| 8 | FLASH | | | 12 | kl (de),a | rl d | |
| 9 | BRIGHT | | | 13 | rlc bc | rl e | |
| 20 | INVERSE | | | 14 | rrc | rl h | |
| 1 | OVER | | | 15 | dec b | rl | |
| 22 | AT | | | 16 | kl a,N | rl (hl) | |
| 23 | TAB | | | 17 | rrc | rl a | |
| 24 | ATTR (JW)(SW) | | | 18 | rrc DIS | rrc b | |
| 25 | POINT (JW)(SW) | | | 19 | rrc hl de | rrc c | |
| 26 | AUTO PAUSE (JW)(SW) | | | 1A | kl a,de | rrc d | |
| 27 | Fill window with character (JW)(SW) | | | 1B | dec de | rrc e | |
| 28 | Set Double Width (JW)(SW) | | | 1C | rrc e | rrc | |
| 29 | Set Font Height (JW) | | | 1D | dec a | rr | |
| 31 | Justification mode (JW)
Set Font Width (SW) | | | 1E | kl e,N | rr (hl) | |
| 32 | Permit embed codes in justif mode (JW)
Redefine Character Set (SW) | | | 1F | rrc | rrc a | |
| 32 | Space | | | 20 | rrc DIS | rrc b | |
| 33 | | | | 21 | kl hl,NN | rrc c | |

| Dec | Character | Control Code | Token | Hex | Z80N Assembler | after CB | after ED |
|-----|-----------|--------------|-------|-----|----------------|-----------|------------|
| 2A | | | | 22 | ld (NN),hl | sla d | |
| 35 | # | | | 23 | rr hl | sla e | swapnt |
| 38 | \$ | | | 24 | rlc h | sla h | mirror a |
| 3 | % | | | 25 | dec h | sla | |
| 38 | & | | | 26 | ld hl,N | sla '(hl) | |
| 39 | | | | 27 | dab | sla a | test N |
| 40 | | | | 28 | rz,DIS | sra b | psla on,b |
| 4 | | | | 29 | add hl,N | sra c | bars de 0 |
| 42 | " | | | 2A | ld hl,(NN) | sra d | hrr da,b |
| 43 | + | | | 2B | dec hl | sra e | bsr' de |
| 44 | | | | 2C | rr | sra h | brlc da,b |
| 45 | | | | 2D | dec | sra | |
| 46 | | | | 2E | ld hl,N | sra '(hl) | |
| 4 | | | | 2F | rpl | sra a | |
| 48 | ^ | | | 30 | rnc,DIS | | mir a e |
| 49 | | | | 31 | ld sp,NN | | add hl,a |
| 50 | 2 | | | 32 | ld (NN),a | | add da,e |
| 5 | 3 | | | 33 | rlc sp | | add hl,a |
| 52 | 4 | | | 34 | inc (hl) | | add hl,NN |
| 53 | 5 | | | 35 | dec (hl) | | add da,NN |
| 54 | E | | | 36 | ld (hl),N | | add da,NN |
| 5 | | | | 37 | scf | | |
| 56 | 8 | | | 38 | jc,DIS | sl b | |
| 5 | 3 | | | 39 | add hl,sp | sl c | |
| 58 | | | | 3A | ld a,(NN) | sl d | |
| 59 | | | | 3B | dec sp | sl e | |
| 60 | < | | | 3C | rr a | sl h | |
| 6 | | | | 3D | dec a | sl | |
| 62 | | | | 3E | ld a,N | sl '(hl) | |
| 63 | 7 | | | 3F | ccf | sl a | |
| 64 | g | | | 40 | ld c,d | sl b | rlt jz |
| 65 | A | | | 41 | ld b,c | sl c | rrl lr h |
| 66 | B | | | 42 | ld c,d | sl d | slc hl,de |
| 67 | C | | | 43 | ld b,b | sl e | ld (NN),br |
| 68 | D | | | 44 | ld c,l | sl h | reg |
| 69 | | | | 45 | ld c, | sl a | rain |
| 70 | F | | | 46 | ld b,(hl) | sl j (hl) | rr d |
| 7 | G | | | 47 | ld b,a | sl "a | ld a |
| 72 | H | | | 48 | ld a,b | sl b | rr c,(c) |
| 73 | | | | 49 | ld c | sl c | out (c) |
| 74 | J | | | 4A | ld c,d | sl d | add hl,b |
| 75 | K | | | 4B | ld c,e | sl e | ld bc,'NN) |
| 76 | L | | | 4C | ld c,h | sl h | |
| 7 | M | | | 4D | ld c, | sl | rgl |
| 78 | N | | | 4E | ld c,(hl) | sl (hl) | |
| 79 | | | | 4F | ld c,a | sl a | ld a |
| 80 | O | | | 50 | ld d,b | sl b | rr d (r) |
| 8 | Q | | | 51 | ld d, | sl c | out (c),G |
| 82 | R | | | 52 | ld d,d | sl d | abc hl,de |
| 83 | S | | | 53 | ld d,e | sl e | ld (NN),de |
| 84 | T | | | 54 | ld d,h | sl h | |
| 86 | U | | | 55 | ld d, | sl | |

Appendix A Character Set Z80N Mnemonics and Control Codes

| Dec | Character | Control Code | Token | Hex | Z80N Assembler | after CB | after ED |
|-----|-----------|--------------|-------|-----|----------------|------------|-------------|
| 86 | V | | | 56 | ld a,(hl) | ld 2,(hl) | em |
| 87 | W | | | 57 | ld a,b | ld 2,a | ld a |
| 88 | X | | | 58 | ld a,l | ld 3,l | ld a,c |
| 89 | Y | | | 59 | ld a,d | ld 3,c | out (c),a |
| 90 | Z | | | 5A | ld a,o | ld 3,d | out (c),a |
| 91 | | | | 5B | ld a,b | ld 3,b | ld (a),(NN) |
| 92 | | | | 5C | ld a,h | ld 3,h | |
| 93 | | | | 5D | ld a, | ld 3, | |
| 94 | ↑ | | | 5E | ld a,(hl) | ld 3,(hl) | em 2 |
| 95 | | | | 5F | ld a,a | ld 3,a | ld a,r |
| 96 | E | | | 60 | ld h,b | ld 4,b | in h,(c) |
| 97 | 3 | | | 61 | ld h,c | ld 4,c | out (c),h |
| 98 | | | | 62 | ld h,d | ld 4,d | ld h,(NN) |
| 99 | | | | 63 | ld h,o | ld 4,o | ld (NN),h |
| 100 | 3 | | | 64 | ld h,r | ld 4,h | |
| 101 | = | | | 65 | ld h, | ld 4, | |
| 102 | | | | 66 | ld h,(hl) | ld 4,(hl) | |
| 103 | G | | | 67 | ld h,a | ld 4,a | ld |
| 104 | h | | | 68 | ld j,b | ld 5,b | in j,c |
| 105 | | | | 69 | ld | ld 5,c | out c, |
| 106 | | | | 6A | ld d | ld 5,d | out h,d |
| 107 | h | | | 6B | ld w | ld 5,b | ld h,(NN) |
| 108 | | | | 6C | ld r | ld 5,h | |
| 109 | m | | | 6D | ld | ld 5, | |
| 110 | n | | | 6E | ld (hl) | ld 5,(hl) | |
| 111 | | | | 6F | ld a | ld 5,a | ld |
| 112 | C | | | 70 | ld (hl),b | ld 6,b | in j,(c) |
| 113 | q | | | 71 | ld (hl),c | ld 6,c | |
| 114 | | | | 72 | ld (hl),d | ld 6,d | ld hl,sp |
| 115 | B | | | 73 | ld (hl),e | ld 6,e | ld (NN),sp |
| 116 | | | | 74 | ld (hl),r | ld 6,h | |
| 117 | u | | | 75 | ld (hl) | ld 6, | |
| 118 | v | | | 76 | ld hl | ld 6,(hl) | |
| 119 | w | | | 77 | ld (hl),a | ld 6,a | |
| 120 | x | | | 78 | ld a,b | ld 7,b | in a,(c) |
| 121 | y | | | 79 | ld a,d | ld 7,c | out (c),a |
| 122 | u | | | 7A | ld a,o | ld 7,d | add hl,sp |
| 123 | | | | 7B | ld a,e | ld 7,e | ld sp,(NN) |
| 124 | | | | 7C | ld a,h | ld 7,h | |
| 125 | | | | 7D | ld a, | ld 7, | |
| 126 | | | | 7E | ld a,(hl) | ld 7,(hl) | |
| 127 | q | | | 7F | ld a,a | ld 7,a | |
| 128 | | | | 80 | add a,b | res 0,a | |
| 129 | ■ | TIME | | 81 | add a,r | res 1,r | |
| 130 | ■ | PRIVATE | | 82 | add a,d | res 0,c | |
| 131 | ■ | IFELSE | | 83 | add a,e | res 0,e | |
| 132 | ■ | ENDIF | | 84 | add a,h | res 0,h | |
| 133 | ■ | EXIT | | 85 | add a, | res 0, | |
| 134 | ■ | REF | | 86 | add a,(hl) | res 1,(hl) | |
| 135 | ■ | PEEK | | 87 | add a,a | res 0,a | |

↑ Displays em if bit indicates E32

| Dec | Character | Control Code | Token | Hex | Z80N Assembler | after CB | after ED |
|-----|-----------|--------------|----------|-----|----------------|------------|------------|
| 36 | ■ | | REG | B3 | adr a,c | res 1,c | |
| 37 | ▀ | | DPOKE | H3 | adr u,c | res | |
| 38 | ▄ | | DPEEK | 5A | adr b,c | res | push NN |
| 39 | └ | | MOD | 8B | adr a,e | res 1 | |
| 40 | ■ | | << | 8C | adr a,n | res 1 | |
| 141 | └ | | >> | 8D | adr a | res | |
| 42 | └ | | UNTIL | 8E | adr a,(hl) | res (hl) | |
| 143 | ■ | | ERROR | BF | adr a,a | res a | |
| 44 | ⓪ | | ON | 90 | siz 1 | res 1 | outline |
| 45 | ⓪ | | DEFPROC | 91 | siz | res 2 | nextreg 1 |
| 46 | ⓪ | | ENDPROC | 92 | siz 1 | res 2 u | nextreg .a |
| 47 | ⓪ | | PROC | 93 | siz e | res 2 e | pushdr |
| 148 | ⓪ | | LOCAL | 94 | siz n | res 2 n | poped |
| 48 | ⓪ | | DRIVER | 95 | siz | res | addr |
| 50 | ⓪ | | WHILE | 96 | siz (hl) | res 2 (hl) | |
| 51 | ⓪ | | RÉPEAT | 97 | siz a | res 2 a | |
| 152 | ⓪ | | ELSE | 98 | sbc a,b | res 2,b | ⓪ (a) |
| 53 | ⓪ | | REMount | 99 | sbc a,r | res 3 n | |
| 54 | ⓪ | | BANK | 9A | str a,d | res 3 d | |
| 55 | ⓪ | | TILE | 9B | str a,e | res 5 e | |
| 56 | ⓪ | | LAYER | 9C | sbc a,h | res 3 n | |
| 157 | ⓪ | | PALETTE | 9D | sbc a | res 3 | |
| 158 | ⓪ | | SPRITE | 9E | sbc a,(hl) | res 3,(hl) | |
| 159 | ⓪ | | PWD | 9F | sbc m,a | res 3,a | |
| 160 | ⓪ | | CD | A0 | and b | res 4,b | ldi |
| 61 | ⓪ | | MKDIR | A1 | and c | res 4,c | cpl |
| 62 | ⓪ | | RANDIR | A2 | and d | res 4,d | ini |
| 63 | ⓪ | | SPECTRUM | A3 | and e | res 4,e | outi |
| 64 | ⓪ | | PLAY | A4 | and h | res 4,n | ldx |
| 65 | ⓪ | | RND | A5 | and | res 4 | ldax |
| 66 | ⓪ | | INKEYS | A6 | and (hl) | res 4 (hl) | |
| 67 | ⓪ | | PI | A7 | and u | res 4 u | |
| 68 | ⓪ | | FN | A8 | xor b | res 5,b | ldi |
| 69 | ⓪ | | POINT | A9 | xor c | res 5,c | cpl |
| 70 | ⓪ | | SCREENS | AA | xor d | res 5,d | ind |
| 171 | ⓪ | | ATTR | AB | xor e | res 5,e | outd |
| 172 | ⓪ | | AT | AC | xor h | res 5,h | lddx |
| 73 | ⓪ | | TAB | AD | xor | res 5 | |
| 74 | ⓪ | | VALS | AE | xor (hl) | res 5 (hl) | |
| 75 | ⓪ | | CODE | AF | xor a | res 5,a | |
| 76 | ⓪ | | VAL | B0 | or b | res 6,b | ldi |
| 77 | ⓪ | | LEN | B1 | or c | res 6 | cpl |
| 178 | ⓪ | | SIN | B2 | or d | res 6,d | ini |
| 179 | ⓪ | | COS | B3 | or e | res 6,e | ptr |
| 180 | ⓪ | | TAN | B4 | or h | res 6,h | lddx |
| 81 | ⓪ | | ASN | B5 | or | res 6 | |
| 82 | ⓪ | | ACS | B6 | or (hl) | res 6,(hl) | |
| 183 | ⓪ | | ATN | B7 | or a | res 6,a | lddx |
| 84 | ⓪ | | LN | B8 | or b | res 7,b | ldx |
| 85 | ⓪ | | EXP | B9 | or c | res 7,c | cpdr |
| 86 | ⓪ | | INT | BA | or d | res 7,d | endi |

Appendix A Character Set Z80M Mnemonics and Control Codes

| Dec | Character | Control Code | Token | Hex | Z80M Assembler | after CB | after ED |
|-----|-----------|--------------|-------|-----|------------------|-----------|----------|
| 187 | | SGR | | BB | cp a | res a | ori |
| 188 | | SGH | | BC | cp h | res r h | ldw |
| 89 | | ABS | | BD | cp | res 7 | |
| 90 | | PEEK | | BE | cp (h) | res 7 (h) | |
| 9 | | IN | | BF | cp a | res 7 a | |
| 92 | | USH | | CD | rel r7 | set a,b | |
| 93 | | STR\$ | | - | pop cr | set r | |
| 94 | | CHR\$ | | C2 | jp nz,NN | set r d | |
| 95 | | NOT | | C3 | jp N/h | set a,b | |
| 96 | | BIN | | C4 | rel r7 NN | set a,r | |
| 97 | | OR | | C5 | push bc | set 7 | |
| 98 | | AND | | C6 | and a,N | set (h) | |
| 99 | | < = | | C7 | res 0 | set a,b | |
| 200 | | > = | | C8 | rel z | set b | |
| 201 | | < > | | C9 | rel | set c | |
| 202 | | LINE | | CA | jp z,NN | set d | |
| 203 | | THEN | | CB | modifying prefix | set a | |
| 204 | | TO | | CC | call z,NN | set h | |
| 205 | | STEP | | CD | call NN | set | |
| 206 | | DEF FN | | CE | add a,N | set (h) | |
| 207 | | CAT | | CF | rst 8 | set a,b | |
| 208 | | FORMAT | | D0 | rel r0 | set a,b | |
| 209 | | MOVE | | D1 | pop de | set a,c | |
| 210 | | ERASE | | D2 | jp nc,NN | set a,d | |
| 211 | | OPEN # | | D3 | out (N),a | set a,b | |
| 212 | | CLOSE # | | D4 | call nc NN | set a,h | |
| 213 | | MERGE | | D5 | push de | set a | |
| 214 | | VERIFY | | D6 | sub N | set a,(h) | |
| 215 | | BEEP | | D7 | rst 16 | set a,b | |
| 216 | | CIRCLE | | D8 | rel c | set a,b | |
| 217 | | INK | | D9 | exx | set a,c | |
| 218 | | PAPER | | DA | jp n,NN | set a,d | |
| 219 | | FLASH | | DB | in a,(h) | set a,b | |
| 220 | | BRIGHT | | DC | call r NN | set a,h | |
| 221 | | INVERSE | | DD | jk prefix* | set a | |
| 222 | | OVER | | DE | adc a,N | set a,(h) | |
| 223 | | OUT | | DF | rst 24 | set a,b | |
| 224 | | LPRINT | | E0 | rel pr | set a,b | |
| 225 | | LIST | | E1 | pop hl | set a,c | |
| 226 | | STOP | | E2 | jp pc,NN | set a,d | |
| 227 | | READ | | E3 | ex (api,N | set a,b | |
| 228 | | DATA | | E4 | call pc,NN | set a,h | |
| 229 | | RESTORE | | E5 | push hl | set a | |
| 230 | | NEW | | E6 | and h | set a,(h) | |
| 231 | | BORDER | | E7 | res 22 | set a,b | |
| 232 | | CONTINUE | | E8 | rel pa | set a,b | |
| 233 | | DIM | | E9 | jp (h) | set a,c | |
| 234 | | REM , | | EA | jp pc,NN | set a,d | |
| 235 | | FOR | | EB | ex de,h | set a,b | |
| 236 | | GO TO | | EC | call pc,N/h | set a,h | |
| 237 | | GO SUB | | ED | modifying prefix | set a | |
| 238 | | INPUT | | EF | xor N | set a (h) | |

| Dec Character | Control Code | Token | Hex | Z80N Assembler | after CB | after ED |
|---------------|--------------|-------|-----|----------------|----------|----------|
| 239 | LOAD | | E1 | rst 40 | set 5,a | |
| 240 | LIST | | F7 | rst 7 | set 4,b | |
| 241 | LET | | F | push ai | set 6,c | |
| 242 | PAUSE | | F2 | jp p,NH | set 5,d | |
| 243 | NEXT | | F3 | di | set 6,e | |
| 244 | POKE | | F4 | call p,NH | set 6,h | |
| 245 | PRINT | | F5 | push ai | set 5 | |
| 246 | PLOT | | F6 | or N | set 6,hi | |
| 247 | RUN | | F7 | rst 48 | set 6,a | |
| 248 | SAVE | | F8 | ret m | set 7,a | |
| 249 | RANDOMIZE | | F9 | ld sp,rl | set 7,c | |
| 250 | IF | | FA | gr m,NH | set 6 | |
| 251 | CLS | | FB | bi | set 7,e | |
| 252 | DRAW | | FC | call m,NH | set 6 | |
| 253 | CLEAR | | FD | iy,para* | set 7 | |
| 254 | RETURN | | FE | cp N | set 7,hi | |
| 255 | COPY | | FF | rst 4d | set 6,d | |

Appendix B Reference

The following sections provide a handy reference of Error Codes and their equivalents, Reports, NextBASIC keywords and functions as well as other information discussed so far in a concise form.

Reports and Error Codes

These appear at the bottom of the screen whenever the computer stops executing some function and explain why it stopped, whether for a natural reason, or because an error occurred.

The report has a brief message explaining what happened and the bank number (not present unless the error occurred in a banked section of program), the line number and statement number, with the one where it stopped. A comment is shown as line 1. Within a line, statement 1 is at the beginning, statement 2 comes after the first colon or THEN and so on. Some of the codes will have a code number or letter so that you can refer to the tables below. There are two types of error reports: General, and Storage System related.

General Errors

The behaviour of **CONTINUE** depends very much on the reports. Normally **CONTINUE** goes to the line and statement specified in the last report, but there are exceptions with reports 0-9 and D.

Below there is a table showing all the reports together with the circumstances they can occur.

| Code | Report | Description | Situation |
|------|-----------------------|--|--|
| 0 | OK | Successful completion of a jump, a line number bigger than any exists (e.g. this report does not change the line and statement jumped to by CONTINUE). | Any |
| | NEXT without FOR | The control variable does not exist or has not been set up by a FOR statement, but there is an ordinary variable with the same name. | NEXT |
| 1 | Variable not found | For a simple variable, this will happen if the variable is used before it has been assigned to in a LET , READ or INPUT statement or loaded into an array or set up in a FOR statement. For a subprogram variable, will happen if the variable also before has been dimensioned in a DIM statement or loaded from a storage device. | Any |
| 2 | Subscript wrong | A subscript is beyond the dimension of the array, or there are the wrong number of subscripts. If the subscript is negative or bigger than 65535, then error B will result. | Subscripted variables
subarrays |
| 3 | Out of memory | There is not enough room in the computer for what you are trying to do. (The computer really seems to be stuck in this state, you may have to flag the command as using up life and then delete a program line or two with the intention of putting them back afterwards) to give yourself room to manoeuvre with, say CLEAR . | LET , INPUT , FOR , DIM , GO SUB , LOAD , MERGE , BANK , PALETTE , SPRITE , AYER , TILE . Some may during expression evaluation. |
| 4 | Out of screen | An INPUT statement has tried to generate more than 23 lines in the lower half of the screen. Also occurs with PRINT AT , 22 , TILE and SPRITE . | INPUT , PRINT AT , SPRITE , TILE |
| 5 | Number too big | Calculations have led to a number greater than about 10 ²⁴ . | Any arithmetic |
| 6 | RETURN without GO SUB | There has been one more RETURN than there were GO SUB s. | RETURN |
| 7 | End of file | | Storage device at operations |
| 8 | STOP statement | After the CONTINUE will not repeat the STOP , but comes on with the statement after. | STOP |

| Code | Report | Description | Situation |
|------|----------------------|---|--|
| A | Invalid argument | The argument for a function is no good for some reason | SQR, LN, ASN, ACS, JSR (with string argument) |
| B | Integer out of range | When an integer is requested, the floating point argument is rounded to the nearest integer. If this is outside a suitable range then error B results. (or any error, see also error 8) | RND%, RANDOMIZE, POKE, DIM, GO TO, GO SUB, LIST, LIST PAUSE, PRINT CHR\$, PEEK, JSR (with numeric argument), PALETTE BANK, SPRITE LAYER, TILE POINT Array, GO, ESS |
| C | Nonsense in BASIC | The text of the (string) argument does not form a valid expression | VAL, VAL\$ |
| D | BREAK/CONT repeats | BREAK was pressed during some nonterminal operation or behavior, or CONT/INUT after his report is normal. That it repeats the statement. Compare with report C. | LOAD, SAVE, VERIFY, MERGE, LPRINT, LIST, COPY. Also when the printer asks 'scroll?' and you type N, SPACE or STOP. |
| E | Out of DATA | You have gone to READ past the end of the DATA list | READ |
| F | Invalid file name | SAVE with name that is empty or unacceptable (see Chapter 20) | SAVE |
| G | No room for line | There is not enough room left in memory to accommodate the new program line | Entering a line into the program |
| H | STOP in INPUT | Some INPUT data started with STOP or, for INPUT, the STOP was pressed. Unlike the case with error 9, after error H, CONTINUE will resume normally by repeating the INPUT statement. | INPUT |
| I | FOR without NEXT | There was a FOR loop to be executed, no times (e.g. FOR n = 1 TO 5) without the corresponding NEXT statement. Or, it may not be found. | FOR |
| J | Invalid I/O device | | Single device I/O operations |
| K | Invalid colour | The number specified is not an appropriate value | INK, PAPER, BORDER, FLASH, BRIGHT, INVERSE, OVER, PALETTE, graphic control characters |
| L | BREAK into program | BREAK pressed, this is detected between two statements. The first statement number (the upper) refers to the statement before BREAK was pressed. The 'CONT' INUT goes to the statement after following for any jumps (e.g. GOTO, GOSUB) does not repeat any statements. | Any |
| M | RAMTOP too good | The number specified for RAMTOP is either too big or too small | CLEAR, BANK, possibly in RUN |
| N | Statement lost | Jump to a statement that no longer exists | RETURN, NEXT, CONTINUE |
| O | Invalid stream | | Storage device, etc. operations |
| P | FN without DEF | An attempt was made to call a function with FN that has not been defined with a matching DEF FN statement | FN |
| Q | Parameter error | Wrong number of arguments, or one of them is the wrong type (string instead of number or vice versa) | FN |
| R | Tape loading error | A file on tape was found but for some reason could not be read in, or would not verify | VERIFY, LOAD or MERGE |
| S | Too many parentheses | Too many parentheses around a repeated character in one of the arguments | PLAY |
| T | Invalid device | The storage device specification does not exist | PLAY |
| U | Invalid note | PLAY came across a note or comment. (didn't recognise it as a comment which was in lower case) | PLAY |
| V | Too big | A parameter for a command is an order of magnitude too big. | PLAY |

STOP occurs normally in extended BASIC as a token that is required for compatibility and does not work with the switch to 40K mode.

| Code | Report | Description | Situation |
|------|----------------------|--|--|
| m | Note out of range | A series of shares or files has taken a note beyond the range of the share chip | PLAY |
| n | Out of range | A parameter for a command is too big or too small. If the error is very large, error results. | PLAY |
| o | Too many tied notes | An attempt was made to tie too many notes together | PLAY |
| | Invalid mode | The mode specified does not exist | LAYER |
| | Direct command error | An attempt was made to execute a command within a situation. This means: do executed a call from the command line or to RUN a procedure definition (DEFPROC) | DEFPROC ERASE
LINE LINE
MERGE BANK
LINE MERGE |
| | Loop error | Occurs in REPEAT, REPEAT UNTIL loops where a matching REPEAT UNTIL REPEAT cannot be found | REPEAT REPEAT
UNTIL WHILE |
| | No DEFPROC | A PROC was found without a matching DEFPROC ENDPROC block | PROC |
| | No ENBIF | An E SEIF was found without a matching ENBIF | IF F SEIF NOIF |
| | No label | A referenced label does not exist | |

Storage Device Related Errors

The following are reports generated by NextZXOS for storage device errors. Those marked in the left-hand column with RIC may be followed by the options **Retry**, **ignore** or **Cancel**?

Some reports may occur with the code(s) shown or without them

| Code | Report | Description |
|------|--------------------------|--|
| e | Already exists | The destination filename or directory already exists. Also occurs when attempting to map a drive letter that is already mapped to another device |
| | Bad file number | An attempt was made to operate on a file which has not been opened. It is unlikely that this error will ever be seen |
| | Bad filename | The filename used does not conform to the filename requirements for the filesystem |
| | Bad parameters | One of the values provided is out of range |
| | Code length error | Trying to load a CODE file from the storage device that is longer than the value given on the LOAD command |
| | Dest. can't be wild | Trying to give a wildcard file specification for the destination file in a COPY command when the source also contains wildcard characters. In this case the destination can only be a drive letter |
| | Dest must be path | The source filename in a COPY command contains wildcard characters, but the destination is only a single filename. In this case the destination can only be a path |
| | Dir full | Unable to add further entries to the directory, or unable to remove a directory because it contains files or subdirectories |
| RIC | Disk changed | The disk in the drive has been changed without properly REMOUNTing. |
| RIC | Disk error | An error has occurred accessing a storage device. If the error persists it may indicate that the device is faulty |
| | Disk full | Saving or copying files to a storage device has used up the free space. The C:\ command can be used to check that there is sufficient free space before attempting such an operation. It may leave a partition like C:\ if there was only space for some of it. This part should be erased, as any attempt to use it will fail |
| | Dot contaminated shot | The error that was trapped by ON ERROR was generated as a dot contaminated file is seen only when ERROR is used to cause the trap error |
| | End of file | An attempt has been made to read a byte past the end-of-file position |
| g,h | File not found | The filename specifies a file that does not exist |
| | Fragmented - use .DEFRAG | The file is split into parts across the disk. Defragment using the .DEFRAG dot command |
| | In use | An attempt has been made to attempt to re-map a file that has files open on it or to access a file that is already open for another purpose |
| | Invalid filename | The filename characters following a colon in a MOVE command is not P, S, or A, or there is more than one character after the colon |
| | Invalid device | The physical device specified does not exist |
| | Invalid drive | A drive letter that does not exist has been specified |
| | Invalid partition | The partition specified does not exist, or is the wrong type |
| | Invalid path | The path specified does not exist |
| | No rename between drives | An attempt has been made to use the MOVE command specifying source and destination filenames that are on different drives |

| | | |
|-----|-------------------|---|
| | No swap partition | An application attempted to access a swap partition but couldn't find one. Install a new swap partition with <code>SWAP</code> and try again. |
| | Not bootable | An attempt has been made to boot a disk image without a boot sector boot program. |
| | Not implemented | An attempt was made to access a facility which isn't available. |
| RIC | Not ready | The storage device was not ready. This usually happens because it has been removed. |
| | Out of handles | There aren't enough handles left to perform the current operation. Unmap a drive and try again. |
| | Partition open | The partition you are trying to delete is already mapped to a drive. |
| RIC | Read only | An attempt has been made to write to a file or storage device which is read-only or has been write-protected. |
| RIC | Seek fail | The device is unable to locate the sector that has been requested. If this error persists it may indicate that the device or disk image is faulty. |
| | Too big | An attempt has been made to write a file that is too large for the filesystem: greater than 64MB for <code>SDOS</code> filesystems, 2GB on <code>ATA</code> or 4GB on <code>ATA2</code> . |
| RIC | Unsuitable media | The device or disk image is formatted in a way that cannot be handled. |
| io | Wrong file type | Trying to <code>LOAD</code> a file of the wrong type (eg trying to load a <code>CODE</code> file as a NextBASIC program). |

NextBASIC Keywords and Functions

The following is a list of all NextBASIC keywords in alphabetical order with a short description regarding their function.

| Keyword | Meaning |
|---|--|
| <code>BANK 346 FORMAT</code> | Reserve banks 346 for use by the RAMdisk again. |
| <code>BANK 346 USE</code> | Allow banks 346 to be used by the BANK command. |
| <code>BANK m COPY o, n</code> | Copy the contents of bank m to bank n. |
| <code>BANK m POKE o, a1...</code> | Double POKE a sequence of comma-separated values starting at offset o in bank m. |
| <code>BANK m ERASE o, n [v]</code> | Fill bank m's optional cycles (all not specified) at optional offset o (if not specified) with value v, or is used if value not specified. |
| <code>BANK m CLEAR</code> | Marks bank m as free for use by other parts of the system. |
| <code>BANK m COPY o, TO n, o2</code> | Copy bytes starting at offset o in bank m to offset o2 in bank n. |
| <code>BANK m GOSUB n</code> | ANSUB line n in bank m. TO SUB line then program from a banked section. Use m: 255. See also: <code>IF</code> , <code>FOR</code> and <code>CALL</code> . |
| <code>BANK m GOTO l</code> | GOTO line l in bank m. To GOTO the main program from a banked section. Use m: 255. |
| <code>BANK m LAYER o(x,y) w(h) TO _app, x,y,w,h jo</code> | Copies pixels from the screen in the current mode from o to o2 in bank m. w(h) is an optional symbol modifier which affects how the data is copied. |
| <code>BANK m LINE x,y</code> | Copies lines x to y inclusive from the main program (or bank m). |
| <code>BANK m LIST [n] (PROC name,...)</code> | List lines (optionally from line n or procedure named name) in bank m. |
| <code>BANK m MERGE</code> | Copy all lines back from bank m into the main program. |
| <code>BANK m POKE o, list</code> | POKE a sequence of comma-separated values starting at offset o in bank m. |
| <code>BANK m PROC name (parameter(s), TO param1)</code> | Call a procedure in bank m. Call a procedure in the main program from a banked section. Use m: 255. See also: <code>IF</code> , <code>FOR</code> . |
| <code>BANK m RESTORE n</code> | Set the DATA pointer to line n in bank m. |
| <code>BANK NOW var</code> | Reserves the next available free bank number and assigns it to the current variable var. |
| <code>BEEP x, y</code> | Sounds a note through the loudspeaker for x seconds at a pitch y semitones above middle C (or below if y is negative). |
| <code>BORDER m</code> | Set the colour of the border of the screen. |
| <code>BRIGHT c</code> | Set brightness of characters subsequently printed: 0 for normal, 1 for bright, 8 for transparent, from 0 to 99, ..., 0 to 8. |
| <code>CAT # Filespec [EXP] [TAB] [ASN]</code> | Produces an alphabetically sorted catalog of files on screen or to an optional stream n from the default drive or according to the optional filespec in standard or EXPANDED form. With the optional TAB and ASN modifiers produces information regarding partitions and drive letter assignments. |
| <code>CD Filespec</code> | Change the current drive and/or directory to the one specified in filespec. |
| <code>CIRCLE x, y, r</code> | Draws an arc of a circle, centre (x,y), radius r. |
| <code>CLEAR n</code> | Deletes all variables freeing the space they occupied. Uses <code>HF</code> , <code>PF</code> and <code>CF</code> reset the <code>CF</code> position to the bottom left-hand corner and clears the NextBASIC Return stack. Optional address n allows to change the RAMTOP to that address. |
| <code>CLOSE #n</code> | Marks stream n as being unformatted on any channel. |
| <code>CLS</code> | Clear Screen. Clears the display of the current layer. |

[illegible]

The following is a list of all NetRAS functions in alphabetical order with a short description regarding their purpose:

[illegible]

| Function | Meaning |
|------------------------------|--|
| BANK PEEK c | Returns the byte at address c if used with the optional BANK the byte at offset c at bank. |
| BANK n PEEKS c,len if | Reads memory eg of of length len stored in the addresses beginning with c and stores it in a string. Reads the string and if it was a set specified string then sets memory with address c, and the offset at BANK reads c, len, ... |
| DI | Returns an approximation of e (3.141592653589793). |
| POINT (x,y) | Returns (x,y) coordinates of point (x,y) on the screen. |
| REG i | Returns state of Regi register. |
| RND (n) | Returns the next pseudorandom number in the range from 0 to 1.0. The next pseudorandom integer number in the range 0 to 10. |
| SCREENS x,y | Returns the character that appears either normally or inverted on the display at line x, column y. |
| SGN x | Signed the sign, 1 for negative, 0 for zero or +1 for positive of x. |
| SGN (x) | Returns a signed integer from integer expression. |
| SIN | Returns the sine of an radians. |
| SQR x | Returns the square root of x. |
| STR\$ x | Returns the string of characters that would be displayed if x were printed. |
| AN x | Returns the larger of x and 0. |
| BANK n USR | calls the machine code subroutine whose starting address is n. With optional BANK goes to same or different bank. It returns the result as a long integer. |
| USR | the address of the subroutine in the user-defined graphics routine. |
| VAL " | evaluates string without its bounding quotes, as a numerical expression. |
| VALS | Evaluates string (without its bounding quotes, as a string expression. |

The Decimal System

Most European languages count using a more or less regular pattern of tens – in English for example although it starts off a bit erratically – soon settles down into regular groups:

twenty twenty one twenty two ... twenty nine
thirty thirty one, thirty two ... thirty nine
forty forty one forty two ... forty nine

This follows from using Arabic numerals which have ten symbols 0–9 in a placeholder system where the position of each digit is multiplied by a power of ten. The reason for using ten as the basis of numbers is that we happen to have ten fingers.

The Binary System

Instead of using the decimal system with ten as its base, computers use a system called binary based on two values 0 and 1. Like humans have ten fingers, computer circuits have two states: low-voltage 0 or 0V or high-voltage 1. The two binary digits are called bits and a bit is either 0 or 1. Computers therefore write 10 to represent 2, 100 to represent 4, 1000 to represent 8, and so on for the powers of 2.

It is customary to pad out binary numbers with leading zeroes so that they always contain at least four bits, called a nibble – for example 0000 0001 0010 0011 representing 0 to 3 decimal. The reason for doing this is that it makes it easy to represent long binary numbers more compactly using hexadecimal, as we will see further below.

Throughout this manual we've written binary numbers either with the suffix **b** (a lower case **b**) or with the prefix **0b** and **Bin** as supported by the NextRASIC integer expression evaluator.

Regardless of how useful it is to write numbers in the way computers understand them, we have the obvious problem of representing them on paper. It's much easier for us to write and understand

05535 + 65534 than **1111111111111111b + 1111111111111110b**

The Hexadecimal System

Binary numbers quickly become unwieldy because even modest quantities require long strings of 0s and 1s to represent them. This is a natural result of only using two symbols to

represent each digit hexadecimal. The term "hex" in short was adopted in easily a more compactly represent binary numbers. Hexadecimal is a base-16 numbering system with 16 symbols. D through 9 are used for the first ten symbols representing decimal values 0-9 and the last six symbols are A B C D E F representing decimal values 10-15. What comes after F? Just as in decimal we write 10 for ten in hexadecimal we write 10 for sixteen since each position is associated with a power of 16.

The reason why hexadecimal is so well suited to representing binary numbers is that sixteen is a power of 2. This means binary digits can be grouped together and directly converted to a hexadecimal digit. Since sixteen is the fourth power of 2, our binary digits (a nibble) can be represented by a single hexadecimal digit. The conversion between binary and hexadecimal can then be done by sight and hexadecimal becomes a quick way to represent large binary quantities as well as an easy way to visualise comparisons.

The table below shows the correspondence between binary, hexadecimal and decimal values.

| Binary | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |
|-------------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| Hexadecimal | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| Decimal | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

To convert hex to binary, change each hex digit into a nibble (four bits) using the table above. Conversely, to convert binary to hex, divide the binary number into nibbles starting on the right and then change each group into the corresponding hex digit.

Throughout this manual we've written hexadecimal numbers suffixed by a lower case letter, typically by \$ as the latter notation is the one supported by the `NextBASIC` Integer Expression evaluator.

Bits, Bytes and Words

The bits inside the computer are mostly grouped into sets of eight — these are called bytes. A single byte can represent any number from 0 to 255 decimal (11111111 binary FFF). A single byte can also represent any character in the ZX Spectrum Next character set. Its value can be written with two hex digits.

Two bytes can be grouped together to make what is called a word. A word can be written using sixteen bits (four hex digits) and represents a number from 0 to 65535 decimal.

A byte is always eight bits, but words vary in length from computer to computer. In Sinclair computers, 16-bit numbers are called words while 32-bit numbers are called long words.

Setting a bit means making a specific bit 1. Resetting a bit means making a specific bit 0. In digital logic, there is also a concept of an active low and active high. It means a signal becomes active when it's 0 or 1 respectively. The Z80r has an `MREQ` or `M1Q` signal, for example. This is an active low signal and to distinguish them from active high signals, we usually write active low signals with a bar over their names. Or prefix them with a forward slash. This means the Z80r indicates a memory cycle by making `MREQ` 0.

Using Binary and Hex in NextBASIC

Our first introduction to binary and hex was in Chapter 7 which introduced Integer Expressions. Chapter 14 introduced the use of the `BIN` keyword. Chapter 16's lower-level use of useful binary was in defining colours with the `PALETTE` keyword while Chapters 23 and 24 went on to introduce binary bitmasks for the `REG` and `OUT` keywords and the memory address space showed the usefulness of hexadecimal.

Really many keyword parameters are binary. As an example, `ATTR` and `RUN ATTR`'s decimal parameters are really decimal translations of the bits that are being set in the computer's memory or the Next Registers that these keywords control.

Appendix C

Machine Personalities

Overview

There is a lot of software originally made for the 48K

logic device called a *Field Programmable Gate Array* (FPGA)

this is what we call *multicore* capability

system software

The Cores and their update procedures

Its main core and a third one for additional cores

computer gets powered up

System New
a new core into the system's flash rom (hence the name Anti Brick)

nally starts the machine

one the second one is reserved only if told so by the release

Specification does not offer additional cores at the time of writing. Its party cores are the responsibility

notes of a **System/Next™** distribution or because your update somehow failed. For example lost power while updating.

The regular core for both Issue 2 and Issue 4 mainboards is contained within a file named **TBBLUE.TBU**. In order to update the flash rom you need to place it on the root folder of your SD Card together with the file containing the firmware **TBBLUE.FW**. Both of these files need to be present for a successful update. Regular operations, however, require only the **TBBLUE.FW** file to be present at all times in the root of your **System/Next™** distribution. Regardless of the update method you need to have them both so make a note for that.

Just placing a **TBBLUE.TBU** file on the root of the card won't update the core. There are additional steps you need to take. Let's examine the two update options below.

Regular Core update

The regular core update method is quite easy. After you've made sure you have the **TBBLUE.FW** and **TBBLUE.TBU** on the root folder of your card, press and hold **U** on your keyboard and while doing that, long² press the **RESET** button. Do not release the **U** key until you see the following screen. Note that if you have a KS- ZX Spectrum Next[™] or a Go computer or a KS1 dev board your Board id will say **ZX Next-Issue 2**.



Fig. 50 Core update screen

Release **U** and then press **Y**. The updater will first calculate the checksum of the core bistream, once it finds everything is OK, it will start upgrading, first erasing the Flash ROM and then, once done successfully, writing the core bistream from **TBBLUE.TBU** in its place. Once the procedure has finished, you will receive an **Updated** Turn the power off and on, message. Remove the power and if using an iDML display, the display cable as well. Wait a few moments and then reconnect everything. The machine should restart with the new core.

AB Core update

If the process failed somehow, or if you're so instructed by the accompanying notes of your **System/Next™** distribution, you can do an **AB** core update to remedy the situation. This is a bit more complicated and it's made so as to avoid entering this mode by mistake.

To enter **AB** core update, you will need to power off your machine, then press and hold the **NMI** and **Drive** buttons together (on the side of the computer) and while doing that, reat-

at the power cable. Wait a few moments then release both keys. You should see the following screen:



Fig. 5 AB core update screen

If the display is blank, press **F3** on the keyboard. Note that due to AB core using the **NM** and **Drive** buttons you cannot press **F3** using the **NMI + 3** shortcut so you must have a PS/2 keyboard for that.

The display could be blank because the AB core works at 60 Hz in vGA mode only so if your display cannot 'lock' onto that mode and you have no PS/2 keyboard to attach, you will need to do a so-called 'blind update'. You can still press **y** and more than likely the update will finish however if you have no display, the preferred method of performing said update is by pressing the **NM** button once which in AB core update is a shortcut, or **y** while the **Drive** button is a shortcut for **n**. If you do perform a "blind update" you should allow the machine adequate time to finish.

Please note that on an issue 4, the average AB update time is 15 minutes from the time you press **y** so allow about 20 minutes before turning the power off.

Multicore (Extra Cores) update

The Extra Cores update deals with the optional third party cores the ZX Spectrum Next accepts. The process is similar with two exceptions. You will need a file called **CORExxx.BIT** where **xxx** is a number from **001** to **031** instead of the **TBBLE.TBU** placed in the root folder of your **System/Next™** distribution and you enter it by pressing and holding **C** instead of **U** while in **NextZXOS**. Every other step is exactly the same. Your 3rd party core will come with instructions on what to do and how to start the core. Generally speaking, files specific to that core go under the **c:/machines/** folder into one subfolder specific to that core. So if, for example, a QL core was released, you would find all pertinent files into **c:/machines/ql/**.

Updating the firmware

In ZX Spectrum Next terminology, *firmware* is the file called **TBBLE.FW** that's located in the root folder of the 5.25" card that holds your **System/Next™** distribution. It is impossible to start the machine without it, as it's a special program that configures all aspects of the machine regardless of personality and core. To update it, you only have to copy the new version over the previous **TBBLE.FW** version. The current FW version is reported on the boot screen. See your *Quik& Start Guide* to see how the core gets reported while booting.

Updating the System/Next™ distribution

Every time a new version of NextZQOS with additional features gets released it gets pushed to the System/Next git repository. Same thing happens with every software, dot-firmware version and core that adds some feature or fixes a bug. A new **System/Next™** will get released in a complete image form only when enough components have been updated as the process is very time consuming and only a large enough update on many components warrants this. So your system updates may be complete (ie. replacing all the components in the system in one go: firmware, core, operating system AND supporting tools, or just partial). You can update your **System/Next™** distribution partially by going to the git repository, at gitlab.com/thismog358/tbblue, downloading the individual components and replacing it in your card. When updating NextZQOS refer to Chapter 19 to find out which files are absolutely required because they all need to be updated together.

The thing said NextZXOS attempts to make things easier for you by using an inbuilt updater program. After downloading the appropriate latest OS update file from www.specnext.com/latestdistro, you need to place it on your SD card's root and then launch the NextZXOS Startup Menu, navigate to More then go to Tools and select Updater. NextZXOS will locate the file and do everything for you.

Alternatively, you can choose to download the entire distribution from gis in one go by selecting the download button on the right top part of the distribution page.

If you do not feel adventurous however, the official home for the System/Next™ distribution is www.speccnext.com/latestdistro/, which also contains links to other forms of the distribution such as complete Sd card images in various sizes for direct burning into SD cards. Alternatively you have the option of purchasing a new SD card with the latest distribution on it from the SpeccNext Ltd store.

Selecting and configuring a personality

When powering up the system, you're presented with the boot screen, where—as we saw in the Quick Start Guide—you're presented with the option of entering the Test Screen or to Press SPACEBAR for Menu.

Pressing **SPACE** (to go back or the option will disappear and booting will continue) will present you with the following screen.



Fig. 52 Paracnave, Sanchon Sinsen

By using the cursor keys and **ENTER** you can select a new personality which will then become your default. If one and all subsequent loads will be your into that selected however!

a personality and pressing E will allow you to configure the specific personality further. Exiting so will present you with another screen.



Fig. 2.4 Configuration Options screen

Note that the screen is broken down into two parts: the top options are always used by Nex7X05 while the bottom part is controlled by Nex7X05's hardware's own enabling and disabling enables hardware configurations according to its needs.

There are a total of 15 personalities available and a few more may become available in a future update depending on the changes which the Native Nex modes come with. The standard 48k ROM and one with the Looking Glass 48K ROM which has the distinctive advanced ROM personality, instead of the standard ROM. However, both these are functionally equivalent and both provide access to all commands in 48K mode.

The Pentagon *28k and T-2048 ones are the most idiosyncratic ones, the first operating only on 50Hz mode and was included to allow access to other custom-built ROMs specially made for custom hardware and the T-2048 being the other. For your personal computer compatible machine.

An important thing to remember is that for compatibility reasons the expansion bus is by default off. This doesn't mean you can plug in interfaces while the machine is working but that you will not have access to external peripherals. This is not exactly all with a series of OUT commands. This is to facilitate the usage of the internal peripherals and the expansion afforded by the Nex's enhanced Z80 processor. All Nex features are available in every mode unless you explicitly turn them off. For example you need to turn off Timex modes via Configuration as above if you don't want them and you must install external ROMs (see relevant section in Chapter 19 on how to do that) in order to access the onboard WMM. Remember that the usage of external peripherals will slow down the machine personality to the 3.5Mhz speed and only the onboard peripherals support the higher speeds. If you don't like Chapter 19 and you know the specific hardware uses, you can enable it yourself with a few easy command sequences.

So that Sinclair RA10 lacks the REG command so as seen in Chapter 22 you will have to issue a series of OUT commands to enable external peripherals. For example to enable a Zx Printer or Alphacom 32 or Timex Sinclair 2040) you will need to give

```
OUT 9275, 136 OUT 9531, 219
OUT 9275, 128 OUT 9531, 128
```

which disables the I/O simulation and immediately slows the expansion bus. You should however disable it afterwards so you can speed the machine up again.

A slightly different example is the following which enables the interface 2. This time the relevant commands are

```
OUT 9275, 128 OUT 9531,0
OUT 9275,2 OUT 9531,1
```

which does things a bit differently. First we select **NextREG 128** (80h, as before but this time we send a value **8** which as you can see from Chapter 29 is an interface. In the table the Expansion Bits after a soft reset and not immediately setting bit 4. The last two OUTs are skippable because the soft reset they initiate can also be done by tapping on your **RESET** button for less than 1 sec.

Troubleshooting

The Next team has taken every possible precaution and measure in order for your ZX Spectrum Next to live for a long time. Inevitably however problems do arise. These are usually not related to the Next and the following paragraphs will hopefully assist you in figuring out quickly what potentially went wrong.

If your screen is blank

- Check that your cables are connected and that your display is on and switched into that input and that your ZX Spectrum Next is powered.
- If the above are working check if you pressed **F3** by mistake or the program you're running has switched modes. (A frequency your monitor doesn't support (eg. 60Hz). Press **NM + 3** to switch frequencies.
- Verify you don't have a monitor that does 60Hz and you switched to Pentagon images which only work at 50Hz. Reset the computer and press **SPACE** upon start to change personalities.
- If you have a DVI monitor verify that your converter is working. Many HDMI or DVI converters do not work with the ZX Spectrum Next. Ask other users at the SpecNext forums for tested converters.
- If you connect your ZX Spectrum Next to a TV or an older CRT monitor via SCART make sure that the line doubler feature is not turned on by mistake. Attempt to remedy by pressing **NM + 2**.

If you see a red screen

- Check the version of the core you're running. If you see a message saying **Core 3.xx.yy required** and update your core.
- Check for a mismatched file versioning of NextZXOS. Prepare the SD card anew.
- If the above are okay, replace your SD card with a new card and repeat the process.

If your PS/2 keyboard is not working

- Check if in configuration mode the PS/2 mode is set to **Keyboard**. **Core v 3.00** and later machines have this setting default to **Mouse**. If you want to use a keyboard, change this in **Keyboard** and if you want to use both you will need to set this mode to Keyboard and purchase a Y-Splitter adapter then plug the keyboard in its appropriate socket.

Other things to look for

Other than the display not being able to support one of the display modes which machine may be in, which is approximately 90% of the cases, the other things to look for is non-reproducible problems. SD card media failures or mis-configuration. As a general guideline we suggest you first study the manual in the relevant sections and if you still cannot figure out the problem, ask for help in SpecNext's forums, our Social Media accounts and the various groups online. If everything else fails, contact SpecNext and we'll try to find you a solution quickly!

Appendix D

The Calculator

The ZX Spectrum Next can be used as a full function calculator.

Selecting the calculator

To use the calculator, call up the *Startup Menu* with **EDIT** and select the *Calculator* option. (If you don't know how to select a menu option, refer back to *Quick Start*.)

The calculator may be selected as soon as the ZX Spectrum Next is switched on.

Alternatively, if you are working on a *NextBASIC* program, you may select the calculator by choosing the *Exit* option from the *Edit/Options Menu*, which returns you to the *Startup Menu*, at which point you can select the *Calculator* option. Note that any *NextBASIC* program which was being worked on, when you selected the calculator, will be remembered and restored when you exit from the calculator and return to *NextBASIC*.

Entering numbers

When you have selected the *Calculator* option, the screen will change to

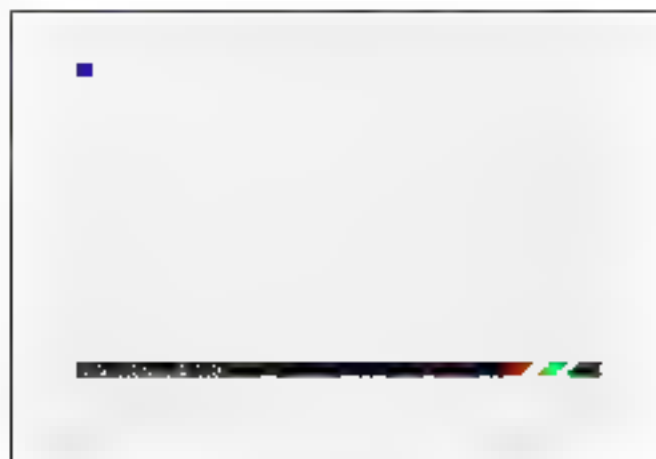


Fig. 54: Calculator Screen

and the ZX Spectrum Next's calculator is ready to accept your first entry. Type in

6+4

As soon as you press **ENTER**, the answer 10 will appear on the next line. (Note that you don't type **=** as you would on a conventional calculator.)

Running total

You will see that the cursor is positioned to the right of the answer, which is a *running total* (like on a conventional calculator). This means that you can simply type in the next operation to be carried out on the running total (without having to type in a whole new calculation). So, with the cursor still positioned to the right of the 10 in the screen, type in

/ 5

and the answer 2 appears.

Using built-in mathematical functions

The ZX Spectrum Next's calculator leverages the power of *NextBASIC* to provide more advanced functions to the user. For example, with the result of the previous operation in place, type in:

```
*PI
```

This produces the result **6.2831853** on the screen. The ZX Spectrum Next has used its built-in π function – all you had to do was type in **PI**. This applies to all the ZX Spectrum Next's mathematical functions. To demonstrate, type in:

```
*ATN 50
```

which will give you the result **9.7648943**.

Editing the screen

To further enhance the calculator's flexibility, you may also edit the contents of the screen. To demonstrate, move the cursor (using the cursor left key) to the beginning of the line and then type in **INT** so that the line reads:

```
INT 9.7648943
```

and as soon as **ENTER** is pressed, the answer **9** will appear. This also demonstrates that the ZX Spectrum Next doesn't have to perform a calculation in order to print the value of an expression. As another example, press **ENTER** and type:

```
1E6
```

which will return the value of that expression. Notice that before you typed in **1E6**, you pressed **ENTER** on its own – this tells the ZX Spectrum Next that you are about to start a new calculation.

Assigning variables

One extremely useful feature of the ZX Spectrum Next's calculator is that it allows you to assign values to variables and then use them in subsequent calculations. This is achieved by using the **LET** statement (unlike *NextBASIC*, the Calculator doesn't yet allow assignments without **LET**). To demonstrate, press **ENTER** and type in the following:

```
LET x=10
```

You must then press **ENTER** twice for the ZX Spectrum Next to accept the variable assignment. Now verify that the variable **x** is being used, by typing:

```
x+50
```

then

```
+x+x
```

If you are using the calculator whilst working on a *NextBASIC* program, then any variables used by the calculator should be chosen so that they do not conflict with those used by the program itself. Note that *NextBASIC* keywords are not allowed to be used as variable names.

User defined functions

Note that if you have set up any user defined functions (using the **DEF FN** statement whilst working on a *NextBASIC* program), you will be able to invoke that function when using the calculator. To illustrate this point, return to *NextBASIC* and type in (for example:


```
9000 DEF FN C(N)=N*N*N
```

which sets up the user defined function `FN C(N)` which returns the cube of `N` (the number you type into the parentheses). Now exit from *NextBASIC* and return to the calculator: you can now use this user defined function as if it were one of the ZX Spectrum Next's own built-in functions. For example, enter

```
FN C(3)
```

and the calculator will print the number 27 (i.e. the cube of 3).

Exiting from the calculator

When you have finished using the calculator, press the `EDIT` key. The screen will change to

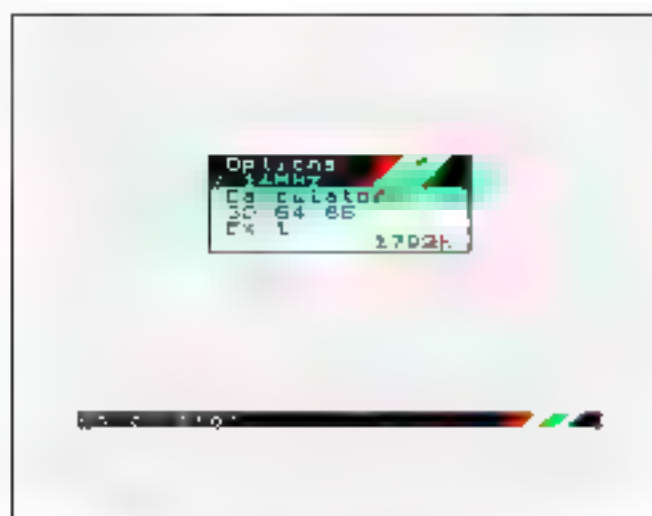


Fig. 55: Calculator Options Menu

Select the *Exit* option to return to the opening menu. If you were working on a *NextBASIC* program before you started using the calculator, then you may return to the program by selecting the *NextBASIC* option. (If you wish to continue using the calculator, then select the *Calculator* option).

Acknowledgements

Production SpecNext Ltd

Henrique Olfiers
Creator Director

Mike Cadwallader
Project Management

Phoebus Dokos Christina Stamatopoulou
Distribution and Logistics

Box

Alfredo Tata
Artwork

Richard Hales, Mike Cadwallader
Copywriters

Manual

Phoebus Dokos
Author

Phoebus Dokos
Artwork (content)

Jonathan M Balts
Artwork (cover)

**Dave Worton, Mike Cadwallader, Alvin Albrecht, Garry Lancaster, David Saphier, Matt Neale,
John Kennedy, Will Stephenson, Rat Mal, FairFightt4, Marc Kloosterman**
Additional & Thraudsourced Editing

Coding

Alvin Albrecht
Spectrum Next Core
with core contributions by Mark Smith and Simon Brattel
Based on a design by Victor Trucco

Garry Lancaster
Firmware, NextZXOS, C/P/M BIOS and Drivers, NextZXOS utilities and NextBASIC

D Rimmon-Souttar
NextPi2

Paul Farrow
ZX80, ZX81 personalities

Geoff Wearmouth
Gosh! Wonderful! & Looking Glass ROMs

Emulation DevSystem

#CSpect
Mike Dailly

Hardware

Alvin Albrecht, ignys
issue 4 board
Based on an original design by Victor Trucco

Software

**Kev Brady, Tim Gilberts, Tony Hoyle, David Saphier, Matt Davies, Robin Verhagen-Guest, D
Rimmon, Garry Lancaster, Gar Biasillo, Simon N Goodwin, Simon Brattel, Neil Mothershead,
Alvin Albrecht, César Hernández Bañó, Peter Helmanovsky, Marco Varesio**

KS 2 Games

David Saphier

Night Knight based on the original by Juan J. Martinez

Michael 'Flash' Ware, Lobo, Space Fractal
Baggers in Space The Detour

Michael 'Flash' Ware, Simon Butler, Space Fractal
Crowley World Tour 2

Michael 'Flash' Ware, Simon Butler, Paul Hesso
Head Over Heels

Hardware Support

Robin Verhagen-Guest

Wi-Fi, nntp, Net

Tim Gilberts

UART, Mouse, RTC, I2C, Additional Wi-Fi support

Marjo Prato

divX, AAC

Industrial Design

Phil Candy, Rick Dickinson

Spectrum Next Case, Keyboard, Packaging

Manufacturing

Dave Worton

Ever Sparkle Technologies

Charlie Hunter, Angela Lennox

Cesatech/Accuralus

Testing

Alvin Albrecht, Jim Bagley, Kev Brady, Mike Cadwallader, Phoebus Dokos, Tim Gilberts
Garry Lancaster, D Rimmon-Soutter, David Saphier
Robin Verhagen-Guest, Simon N Goodwin

Special Thanks

Alvin Albrecht, Gari Blasillo, Kev Brady, Mike Dally, Matt Davies, Tim Gilberts,
Garry Lancaster, Lampros Potamianos, D Rimmon-Soutter, David Saphier
Lyndon J Sharp, Michael 'Flash' Ware

For their unwavering drive and dedication in creating, enhancing, guiding and pushing the Spectrum Next forwards

Our Kickstarter backers and shop purchasers

For their belief and patience and for making the whole endeavour possible

Additional Thanks

Manuel Fernandez Higuera

For creating the first fully compatible Spectrum Next board, the N-GP, as well as testing and incorporating improvements to the platform

Antonio Viliena

For creating the gomaDOS+ a ZX DOS board compatible with the Spectrum Next

Don Superio

For creating the first ultra-mini compatible Next Board, the X-Berry π

Evgeniy Barskiy, Dimitr Ponomarev

Enhanced/LA ideas behind extra colour modes on layers 0 and

David Banks

BRQ Core

Table of Contents

| | | | |
|--------------------------------------|-----|------------------------------|-----|
| Chapter 1 Basic Programming Concepts | 1 | Chapter 9 Standard Functions | 64 |
| 1.1 Introduction | 1 | 9.1 Arrays | 6 |
| 1.2 Variables | 2 | 9.2 Strings | 7 |
| 1.3 Constants | 3 | 9.3 Indirects | 7 |
| 1.4 Arithmetic Operators | 4 | 9.4 Pointer Set | 14 |
| 1.5 Relational Operators | 5 | 9.5 Bitwise Operators | 15 |
| 1.6 Logical Operators | 6 | 9.6 Miscellaneous | 16 |
| 1.7 Decision Making | 7 | 9.7 Miscellaneous | 17 |
| 1.8 Looping | 8 | 9.8 Miscellaneous | 18 |
| 1.9 Functions | 9 | 9.9 Miscellaneous | 19 |
| 1.10 Preprocessor | 10 | 9.10 Miscellaneous | 20 |
| Chapter 2 Data Types | 11 | 9.11 Miscellaneous | 21 |
| 2.1 Data Types | 11 | 9.12 Miscellaneous | 22 |
| 2.2 Operators | 12 | 9.13 Miscellaneous | 23 |
| 2.3 Expressions | 13 | 9.14 Miscellaneous | 24 |
| 2.4 Statements | 14 | 9.15 Miscellaneous | 25 |
| 2.5 Control Flow | 15 | 9.16 Miscellaneous | 26 |
| 2.6 Data Flow | 16 | 9.17 Miscellaneous | 27 |
| 2.7 Memory Management | 17 | 9.18 Miscellaneous | 28 |
| 2.8 File Handling | 18 | 9.19 Miscellaneous | 29 |
| 2.9 Networking | 19 | 9.20 Miscellaneous | 30 |
| 2.10 Database | 20 | 9.21 Miscellaneous | 31 |
| 2.11 Security | 21 | 9.22 Miscellaneous | 32 |
| 2.12 Graphics | 22 | 9.23 Miscellaneous | 33 |
| 2.13 Audio | 23 | 9.24 Miscellaneous | 34 |
| 2.14 Video | 24 | 9.25 Miscellaneous | 35 |
| 2.15 Animation | 25 | 9.26 Miscellaneous | 36 |
| 2.16 Game Development | 26 | 9.27 Miscellaneous | 37 |
| 2.17 Mobile Development | 27 | 9.28 Miscellaneous | 38 |
| 2.18 Cloud Development | 28 | 9.29 Miscellaneous | 39 |
| 2.19 Big Data | 29 | 9.30 Miscellaneous | 40 |
| 2.20 Internet of Things | 30 | 9.31 Miscellaneous | 41 |
| 2.21 Artificial Intelligence | 31 | 9.32 Miscellaneous | 42 |
| 2.22 Blockchain | 32 | 9.33 Miscellaneous | 43 |
| 2.23 Quantum Computing | 33 | 9.34 Miscellaneous | 44 |
| 2.24 Nanotechnology | 34 | 9.35 Miscellaneous | 45 |
| 2.25 Space Technology | 35 | 9.36 Miscellaneous | 46 |
| 2.26 Biotechnology | 36 | 9.37 Miscellaneous | 47 |
| 2.27 Environmental Technology | 37 | 9.38 Miscellaneous | 48 |
| 2.28 Energy Technology | 38 | 9.39 Miscellaneous | 49 |
| 2.29 Transportation Technology | 39 | 9.40 Miscellaneous | 50 |
| 2.30 Agriculture Technology | 40 | 9.41 Miscellaneous | 51 |
| 2.31 Healthcare Technology | 41 | 9.42 Miscellaneous | 52 |
| 2.32 Manufacturing Technology | 42 | 9.43 Miscellaneous | 53 |
| 2.33 Construction Technology | 43 | 9.44 Miscellaneous | 54 |
| 2.34 Retail Technology | 44 | 9.45 Miscellaneous | 55 |
| 2.35 Financial Technology | 45 | 9.46 Miscellaneous | 56 |
| 2.36 Media Technology | 46 | 9.47 Miscellaneous | 57 |
| 2.37 Telecommunications Technology | 47 | 9.48 Miscellaneous | 58 |
| 2.38 Aerospace Technology | 48 | 9.49 Miscellaneous | 59 |
| 2.39 Defense Technology | 49 | 9.50 Miscellaneous | 60 |
| 2.40 Space Exploration Technology | 50 | 9.51 Miscellaneous | 61 |
| 2.41 Robotics Technology | 51 | 9.52 Miscellaneous | 62 |
| 2.42 Nanotechnology | 52 | 9.53 Miscellaneous | 63 |
| 2.43 Quantum Computing | 53 | 9.54 Miscellaneous | 64 |
| 2.44 Nanotechnology | 54 | 9.55 Miscellaneous | 65 |
| 2.45 Space Technology | 55 | 9.56 Miscellaneous | 66 |
| 2.46 Biotechnology | 56 | 9.57 Miscellaneous | 67 |
| 2.47 Environmental Technology | 57 | 9.58 Miscellaneous | 68 |
| 2.48 Energy Technology | 58 | 9.59 Miscellaneous | 69 |
| 2.49 Transportation Technology | 59 | 9.60 Miscellaneous | 70 |
| 2.50 Agriculture Technology | 60 | 9.61 Miscellaneous | 71 |
| 2.51 Healthcare Technology | 61 | 9.62 Miscellaneous | 72 |
| 2.52 Manufacturing Technology | 62 | 9.63 Miscellaneous | 73 |
| 2.53 Construction Technology | 63 | 9.64 Miscellaneous | 74 |
| 2.54 Retail Technology | 64 | 9.65 Miscellaneous | 75 |
| 2.55 Financial Technology | 65 | 9.66 Miscellaneous | 76 |
| 2.56 Media Technology | 66 | 9.67 Miscellaneous | 77 |
| 2.57 Telecommunications Technology | 67 | 9.68 Miscellaneous | 78 |
| 2.58 Aerospace Technology | 68 | 9.69 Miscellaneous | 79 |
| 2.59 Defense Technology | 69 | 9.70 Miscellaneous | 80 |
| 2.60 Space Exploration Technology | 70 | 9.71 Miscellaneous | 81 |
| 2.61 Robotics Technology | 71 | 9.72 Miscellaneous | 82 |
| 2.62 Nanotechnology | 72 | 9.73 Miscellaneous | 83 |
| 2.63 Quantum Computing | 73 | 9.74 Miscellaneous | 84 |
| 2.64 Nanotechnology | 74 | 9.75 Miscellaneous | 85 |
| 2.65 Space Technology | 75 | 9.76 Miscellaneous | 86 |
| 2.66 Biotechnology | 76 | 9.77 Miscellaneous | 87 |
| 2.67 Environmental Technology | 77 | 9.78 Miscellaneous | 88 |
| 2.68 Energy Technology | 78 | 9.79 Miscellaneous | 89 |
| 2.69 Transportation Technology | 79 | 9.80 Miscellaneous | 90 |
| 2.70 Agriculture Technology | 80 | 9.81 Miscellaneous | 91 |
| 2.71 Healthcare Technology | 81 | 9.82 Miscellaneous | 92 |
| 2.72 Manufacturing Technology | 82 | 9.83 Miscellaneous | 93 |
| 2.73 Construction Technology | 83 | 9.84 Miscellaneous | 94 |
| 2.74 Retail Technology | 84 | 9.85 Miscellaneous | 95 |
| 2.75 Financial Technology | 85 | 9.86 Miscellaneous | 96 |
| 2.76 Media Technology | 86 | 9.87 Miscellaneous | 97 |
| 2.77 Telecommunications Technology | 87 | 9.88 Miscellaneous | 98 |
| 2.78 Aerospace Technology | 88 | 9.89 Miscellaneous | 99 |
| 2.79 Defense Technology | 89 | 9.90 Miscellaneous | 100 |
| 2.80 Space Exploration Technology | 90 | 9.91 Miscellaneous | 101 |
| 2.81 Robotics Technology | 91 | 9.92 Miscellaneous | 102 |
| 2.82 Nanotechnology | 92 | 9.93 Miscellaneous | 103 |
| 2.83 Quantum Computing | 93 | 9.94 Miscellaneous | 104 |
| 2.84 Nanotechnology | 94 | 9.95 Miscellaneous | 105 |
| 2.85 Space Technology | 95 | 9.96 Miscellaneous | 106 |
| 2.86 Biotechnology | 96 | 9.97 Miscellaneous | 107 |
| 2.87 Environmental Technology | 97 | 9.98 Miscellaneous | 108 |
| 2.88 Energy Technology | 98 | 9.99 Miscellaneous | 109 |
| 2.89 Transportation Technology | 99 | 9.100 Miscellaneous | 110 |
| 2.90 Agriculture Technology | 100 | 9.101 Miscellaneous | 111 |
| 2.91 Healthcare Technology | 101 | 9.102 Miscellaneous | 112 |
| 2.92 Manufacturing Technology | 102 | 9.103 Miscellaneous | 113 |
| 2.93 Construction Technology | 103 | 9.104 Miscellaneous | 114 |
| 2.94 Retail Technology | 104 | 9.105 Miscellaneous | 115 |
| 2.95 Financial Technology | 105 | 9.106 Miscellaneous | 116 |
| 2.96 Media Technology | 106 | 9.107 Miscellaneous | 117 |
| 2.97 Telecommunications Technology | 107 | 9.108 Miscellaneous | 118 |
| 2.98 Aerospace Technology | 108 | 9.109 Miscellaneous | 119 |
| 2.99 Defense Technology | 109 | 9.110 Miscellaneous | 120 |
| 2.100 Space Exploration Technology | 110 | 9.111 Miscellaneous | 121 |
| 2.101 Robotics Technology | 111 | 9.112 Miscellaneous | 122 |
| 2.102 Nanotechnology | 112 | 9.113 Miscellaneous | 123 |
| 2.103 Quantum Computing | 113 | 9.114 Miscellaneous | 124 |
| 2.104 Nanotechnology | 114 | 9.115 Miscellaneous | 125 |
| 2.105 Space Technology | 115 | 9.116 Miscellaneous | 126 |
| 2.106 Biotechnology | 116 | 9.117 Miscellaneous | 127 |
| 2.107 Environmental Technology | 117 | 9.118 Miscellaneous | 128 |
| 2.108 Energy Technology | 118 | 9.119 Miscellaneous | 129 |
| 2.109 Transportation Technology | 119 | 9.120 Miscellaneous | 130 |
| 2.110 Agriculture Technology | 120 | 9.121 Miscellaneous | 131 |
| 2.111 Healthcare Technology | 121 | 9.122 Miscellaneous | 132 |
| 2.112 Manufacturing Technology | 122 | 9.123 Miscellaneous | 133 |
| 2.113 Construction Technology | 123 | 9.124 Miscellaneous | 134 |
| 2.114 Retail Technology | 124 | 9.125 Miscellaneous | 135 |
| 2.115 Financial Technology | 125 | 9.126 Miscellaneous | 136 |
| 2.116 Media Technology | 126 | 9.127 Miscellaneous | 137 |
| 2.117 Telecommunications Technology | 127 | 9.128 Miscellaneous | 138 |
| 2.118 Aerospace Technology | 128 | 9.129 Miscellaneous | 139 |
| 2.119 Defense Technology | 129 | 9.130 Miscellaneous | 140 |
| 2.120 Space Exploration Technology | 130 | 9.131 Miscellaneous | 141 |
| 2.121 Robotics Technology | 131 | 9.132 Miscellaneous | 142 |
| 2.122 Nanotechnology | 132 | 9.133 Miscellaneous | 143 |
| 2.123 Quantum Computing | 133 | 9.134 Miscellaneous | 144 |
| 2.124 Nanotechnology | 134 | 9.135 Miscellaneous | 145 |
| 2.125 Space Technology | 135 | 9.136 Miscellaneous | 146 |
| 2.126 Biotechnology | 136 | 9.137 Miscellaneous | 147 |
| 2.127 Environmental Technology | 137 | 9.138 Miscellaneous | 148 |
| 2.128 Energy Technology | 138 | 9.139 Miscellaneous | 149 |
| 2.129 Transportation Technology | 139 | 9.140 Miscellaneous | 150 |
| 2.130 Agriculture Technology | 140 | 9.141 Miscellaneous | 151 |
| 2.131 Healthcare Technology | 141 | 9.142 Miscellaneous | 152 |
| 2.132 Manufacturing Technology | 142 | 9.143 Miscellaneous | 153 |
| 2.133 Construction Technology | 143 | 9.144 Miscellaneous | 154 |
| 2.134 Retail Technology | 144 | 9.145 Miscellaneous | 155 |
| 2.135 Financial Technology | 145 | 9.146 Miscellaneous | 156 |
| 2.136 Media Technology | 146 | 9.147 Miscellaneous | 157 |
| 2.137 Telecommunications Technology | 147 | 9.148 Miscellaneous | 158 |
| 2.138 Aerospace Technology | 148 | 9.149 Miscellaneous | 159 |
| 2.139 Defense Technology | 149 | 9.150 Miscellaneous | 160 |
| 2.140 Space Exploration Technology | 150 | 9.151 Miscellaneous | 161 |
| 2.141 Robotics Technology | 151 | 9.152 Miscellaneous | 162 |
| 2.142 Nanotechnology | 152 | 9.153 Miscellaneous | 163 |
| 2.143 Quantum Computing | 153 | 9.154 Miscellaneous | 164 |
| 2.144 Nanotechnology | 154 | 9.155 Miscellaneous | 165 |
| 2.145 Space Technology | 155 | 9.156 Miscellaneous | 166 |
| 2.146 Biotechnology | 156 | 9.157 Miscellaneous | 167 |
| 2.147 Environmental Technology | 157 | 9.158 Miscellaneous | 168 |
| 2.148 Energy Technology | 158 | 9.159 Miscellaneous | 169 |
| 2.149 Transportation Technology | 159 | 9.160 Miscellaneous | 170 |
| 2.150 Agriculture Technology | 160 | 9.161 Miscellaneous | 171 |
| 2.151 Healthcare Technology | 161 | 9.162 Miscellaneous | 172 |
| 2.152 Manufacturing Technology | 162 | 9.163 Miscellaneous | 173 |
| 2.153 Construction Technology | 163 | 9.164 Miscellaneous | 174 |
| 2.154 Retail Technology | 164 | 9.165 Miscellaneous | 175 |
| 2.155 Financial Technology | 165 | 9.166 Miscellaneous | 176 |
| 2.156 Media Technology | 166 | 9.167 Miscellaneous | 177 |
| 2.157 Telecommunications Technology | 167 | 9.168 Miscellaneous | 178 |
| 2.158 Aerospace Technology | 168 | 9.169 Miscellaneous | 179 |
| 2.159 Defense Technology | 169 | 9.170 Miscellaneous | 180 |
| 2.160 Space Exploration Technology | 170 | 9.171 Miscellaneous | 181 |
| 2.161 Robotics Technology | 171 | 9.172 Miscellaneous | 182 |
| 2.162 Nanotechnology | 172 | 9.173 Miscellaneous | 183 |
| 2.163 Quantum Computing | 173 | 9.174 Miscellaneous | 184 |
| 2.164 Nanotechnology | 174 | 9.175 Miscellaneous | 185 |
| 2.165 Space Technology | 175 | 9.176 Miscellaneous | 186 |
| 2.166 Biotechnology | 176 | 9.177 Miscellaneous | 187 |
| 2.167 Environmental Technology | 177 | 9.178 Miscellaneous | 188 |
| 2.168 Energy Technology | 178 | 9.179 Miscellaneous | 189 |
| 2.169 Transportation Technology | 179 | 9.180 Miscellaneous | 190 |
| 2.170 Agriculture Technology | 180 | 9.181 Miscellaneous | 191 |
| 2.171 Healthcare Technology | 181 | 9.182 Miscellaneous | 192 |
| 2.172 Manufacturing Technology | 182 | 9.183 Miscellaneous | 193 |
| 2.173 Construction Technology | 183 | 9.184 Miscellaneous | 194 |
| 2.174 Retail Technology | 184 | 9.185 Miscellaneous | 195 |
| 2.175 Financial Technology | 185 | 9.186 Miscellaneous | 196 |
| 2.176 Media Technology | 186 | 9.187 Miscellaneous | 197 |
| 2.177 Telecommunications Technology | 187 | 9.188 Miscellaneous | 198 |
| 2.178 Aerospace Technology | 188 | 9.189 Miscellaneous | 199 |
| 2.179 Defense Technology | 189 | 9.190 Miscellaneous | 200 |
| 2.180 Space Exploration Technology | 190 | 9.191 Miscellaneous | 201 |
| 2.181 Robotics Technology | 191 | 9.192 Miscellaneous | 202 |
| 2.182 Nanotechnology | 192 | 9.193 Miscellaneous | 203 |
| 2.183 Quantum Computing | 193 | 9.194 Miscellaneous | 204 |
| 2.184 Nanotechnology | 194 | 9.195 Miscellaneous | 205 |
| 2.185 Space Technology | 195 | 9.196 Miscellaneous | 206 |
| 2.186 Biotechnology | 196 | 9.197 Miscellaneous | 207 |
| 2.187 Environmental Technology | 197 | 9.198 Miscellaneous | 208 |
| 2.188 Energy Technology | 198 | 9.199 Miscellaneous | 209 |
| 2.189 Transportation Technology | 199 | 9.200 Miscellaneous | 210 |
| 2.190 Agriculture Technology | 200 | 9.201 Miscellaneous | 211 |
| 2.191 Healthcare Technology | 201 | 9.202 Miscellaneous | 212 |
| 2.192 Manufacturing Technology | 202 | 9.203 Miscellaneous | 213 |
| 2.193 Construction Technology | 203 | 9.204 Miscellaneous | 214 |
| 2.194 Retail Technology | 204 | 9.205 Miscellaneous | 215 |
| 2.195 Financial Technology | 205 | 9.206 Miscellaneous | 216 |
| 2.196 Media Technology | 206 | 9.207 Miscellaneous | 217 |
| 2.197 Telecommunications Technology | 207 | 9.208 Miscellaneous | 218 |
| 2.198 Aerospace Technology | 208 | 9.209 Miscellaneous | 219 |
| 2.199 Defense Technology | 209 | 9.210 Miscellaneous | 220 |
| 2.200 Space Exploration Technology | 210 | 9.211 Miscellaneous | 221 |
| 2.201 Robotics Technology | 211 | 9.212 Miscellaneous | 222 |
| 2.202 Nanotechnology | 212 | 9.213 Miscellaneous | 223 |
| 2.203 Quantum Computing | 213 | 9.214 Miscellaneous | 224 |
| 2.204 Nanotechnology | 214 | 9.215 Miscellaneous | 225 |
| 2.205 Space Technology | 215 | 9.216 Miscellaneous | 226 |
| 2.206 Biotechnology | 216 | 9.217 Miscellaneous | 227 |
| 2.207 Environmental Technology | 217 | 9.218 Miscellaneous | 228 |
| 2.208 Energy Technology | 218 | 9.219 Miscellaneous | 229 |
| 2.209 Transportation Technology | 219 | 9.220 Miscellaneous | 230 |
| 2.210 Agriculture Technology | 220 | 9.221 Miscellaneous | 231 |
| 2.211 Healthcare Technology | 221 | 9.222 Miscellaneous | 232 |
| 2.212 Manufacturing Technology | 222 | 9.223 Miscellaneous | 233 |
| 2.213 Construction Technology | 223 | 9.224 Miscellaneous | 234 |
| 2.214 Retail Technology | 224 | 9.225 Miscellaneous | 235 |
| 2.215 Financial Technology | 225 | 9.226 Miscellaneous | 236 |
| 2.216 Media Technology | 226 | 9.227 Miscellaneous | 237 |
| 2.217 Telecommunications Technology | 227 | 9.228 Miscellaneous | 238 |
| 2.218 Aerospace Technology | 228 | 9.229 Miscellaneous | 239 |
| 2.219 Defense Technology | 229 | 9.230 Miscellaneous | 240 |
| 2.220 Space Exploration Technology | 230 | 9.231 Miscellaneous | 241 |
| 2.221 Robotics Technology | 231 | 9.232 Miscellaneous | 242 |
| 2.222 Nanotechnology | 232 | 9.233 Miscellaneous | 243 |
| 2.223 Quantum Computing | 233 | 9.234 Miscellaneous | 244 |
| 2.224 Nanotechnology | 234 | 9.235 Miscellaneous | 245 |
| 2.225 Space Technology | 235 | 9.236 Miscellaneous | 246 |
| 2.226 Biotechnology | 236 | 9.237 Miscellaneous | 247 |
| 2.227 Environmental Technology | 237 | 9.238 Miscellaneous | 248 |
| 2.228 Energy Technology | 238 | 9.239 Miscellaneous | 249 |
| 2.229 Transportation Technology | 239 | 9.240 Miscellaneous | 250 |
| 2.230 Agriculture Technology | 240 | 9.241 Miscellaneous | 251 |
| 2.231 Healthcare Technology | 241 | 9.242 Miscellaneous | 252 |
| 2.232 Manufacturing Technology | 242 | 9.243 Miscellaneous | 253 |
| 2.233 Construction Technology | 243 | 9.244 Miscellaneous | 254 |
| 2.234 Retail Technology | 244 | 9.245 Miscellaneous | 255 |
| 2.235 Financial Technology | 245 | 9.246 Miscellaneous | 256 |
| 2.236 Media Technology | 246 | 9.247 Miscellaneous | 257 |
| 2.237 Telecommunications Technology | 247 | 9.248 Miscellaneous | 258 |
| 2.238 Aerospace Technology | 248 | 9.249 Miscellaneous | 259 |
| 2.239 Defense Technology | 249 | 9.250 Miscellaneous | 260 |
| 2.240 Space Exploration Technology | 250 | 9.251 Miscellaneous | 261 |
| 2.241 Robotics Technology | 251 | 9.252 Miscellaneous | 262 |
| 2.242 Nanotechnology | 252 | 9.253 Miscellaneous | 263 |
| 2.243 Quantum Computing | 253 | 9.254 Miscellaneous | 264 |
| 2.244 Nanotechnology | 254 | 9.255 Miscellaneous | 265 |
| 2.245 Space Technology | 255 | 9.2. | |

Chapter 18 Sound and Music

Basic sounds with the BEB- command

Chapter 18 Sound and Music

Basic sounds with the BEB- command

Chapter 18 Sound and Music

Basic sounds with the BEB- command

Chapter 18 Sound and Music

Basic sounds with the BEB- command

Chapter 18 Sound and Music

Basic sounds with the BEB- command

Chapter 18 Sound and Music

Basic sounds with the BEB- command

Chapter 18 Sound and Music

Basic sounds with the BEB- command

Chapter 18 Sound and Music

Basic sounds with the BEB- command

Chapter 18 Sound and Music

Basic sounds with the BEB- command

Chapter 18 Sound and Music

Basic sounds with the BEB- command

Chapter 19 Sound and Music

Basic sounds with the BEB- command

Chapter 19 Sound and Music

Basic sounds with the BEB- command

Chapter 19 Sound and Music

Basic sounds with the BEB- command

Chapter 19 Sound and Music

Basic sounds with the BEB- command

Chapter 19 Sound and Music

Basic sounds with the BEB- command

Chapter 19 Sound and Music

Basic sounds with the BEB- command

Chapter 19 Sound and Music

Basic sounds with the BEB- command

Chapter 19 Sound and Music

Basic sounds with the BEB- command

Chapter 19 Sound and Music

Basic sounds with the BEB- command

Chapter 19 Sound and Music

Basic sounds with the BEB- command

Chapter 19 Sound and Music

Basic sounds with the BEB- command

Chapter 19 Sound and Music

Basic sounds with the BEB- command

Table Of Contents

| | |
|---|------------|
| Upgrading the hardware | 289 |
| upgrading the System/PowerPC distribution | 290 |
| Switching from Macintosh to Macintosh | 300 |
| Troubleshooting | 302 |
| What things to look for | 303 |
| Appendix Q- The Calculator | 303 |
| Getting the calculator | 303 |
| Entering numbers | 303 |
| Running total | 303 |
| Using mathematical functions | 304 |
| Editing the screen | 304 |
| Assigning variables | 304 |
| Getting and setting | 304 |
| Editing on the calculator | 305 |
| Acknowledgements | 307 |
| Table Of Contents | 309 |
| Index | 312 |
| Technical Specifications | 317 |
| Issue 2 | 2 |
| Issue 4 | 9 |

Index

| | | | | | | |
|-----------------------------------|---|---------------------------|-----------------|-----------------|------------------------|---|
| D | | | | | graphics symbols | 74 |
| DA08 | | | | 155 | GRB | 95 |
| DA A | | 33,35,36,67 | 70 | 28 | H | |
| DA A (function) | | | | 36 | hardware scrolling | 40 |
| Decimal | | | | 4 | hexadecimal | 4 |
| DEF FN | | | 58,57,61 | | HColour | 81, 99, 100 |
| DEFPROC | | 29,30,3 | 53 | | HRes | 81, 84, 89, 98, 99, 100 |
| degrees | + | + | + | 53 | horizontal coordinates | 98 |
| DELETE | | | | 74 | horizontal resolution | 96 |
| DIM | | | | 87, 88, 89 | horizontal size | 96 |
| DIM function | | | | 70 | Hz | 2 |
| dimension | | | | 68 | | |
| DISP FILE | | | 9 | 98, 99, 100 | IPS | 55 |
| DISP FILE | | | | 97, 98, 100 | IF | 81, 7, 20, 71 |
| DISP FILE? | | | | 97, 98, 99, 100 | IN | 54, 53, 60, 100 |
| dot command | + | | | 22 | INK | 75, 89, 100, 105, 106, 109, 115, 120 |
| DPEEK | | | | 69 | INKEY\$ | 83, 125 |
| DRAW | | 86 | 3 | 14, 15, 16 | INPUT | 81, 21, 28, 42, 59, 67, 68, 84, 90, 91, 93, 106, 115, 116 |
| E | | | | | INPL T function | 94 |
| EDIT | | | | 17 | INPL T item | 90, 91 |
| Edt menu | | | | 89 | INPL T INE | 8 |
| ELSE | | 6 | 7, 8, 9, 20, 36 | | INPUT n | 83 |
| ELSE IF | | | | 18 | INT | 56, 57, 64 |
| END IF | | | | 8, 20 | INT | 42, 58 |
| ENDPROC | | 9, 24, 25, 30, 31, 32, 42 | | | Integer arithmetic | 47 |
| EnhancedULA | | 81, 82, 100, 101, 106 | 99, 115 | | integer array | 67, 68 |
| ENTER | | | 65, 91 | 25 | Integer Expressions | 37 |
| Err | | | | 90 | interleaved storage | 96 |
| even-tempered tuning | + | | | 48 | INVERSE | 97, 5, 7, 8, 20 |
| EXIT | + | | | 19, 23, 24 | K | |
| EXP | | | | 60, 6 | K Invid Colour | 103 |
| Extension | | | | 42 | keyboard joystick | 94 |
| Expressions | | | | 39 | | |
| Extended colour attribute display | | | | 98 | L | |
| F | | | | | Labels | 8 |
| false | | | | 71, 72 | LAYER | 81, 89, 94, 100, 103, 104, 106, 113, 121 |
| FLASH | | 90, 102, 105, 106, 109 | 5, 26 | | Layer 0 | 81, 82, 84, 95, 98, 99, 90, 91, 106 |
| FN | + | | | 58, 57 | Layer | 81, 84, 85 |
| FOR | | 21, 22, 23, 24, 42, 114 | | | LAYER ,0 | 86, 92 |
| FPGA | | | | 28 | LAYER 1,1 | 81, 88 |
| FRAMES | | | | 122 | LAYER 2 | 89, 90 |
| Full Ink Mode | | | | 01 | LAYER 3 | 110 |
| function | | | | 44, 52 | Layer 2 ++ ++ + | 81, 83, 86, 97, 102, 5 |
| G | | | | | Layer 3 | 80, 81, 83, 97 |
| G R B1 | | | | 95 | LAYER AT ++ + | 140 |
| G3R3B2 | | | | 95 | LAYER CLEAR | 105, 8 |
| G4 SUB | | | | 27 | LAYER DIM | 119 |
| G0 TO | | 17, 18, 19, 20, 23, 27 | 85 | | LAYER ERASE + + | 18, 18 |
| GPIO | + | | | 55 | LAYER OVER | 103, 104, 118 |
| GRAPHICS | | | | 74 | LAYER PALETTE | 100, 108, 110, 115 |
| graphics modes | | | | 81 | LAYER PALETTE BANK | 109 |

| | | | | | |
|--------------------------|----|----|----|--------|----------------|
| LINE | | | | | 33 |
| LEFT | | | | | 5,16,42,81,117 |
| LINE input | ++ | ++ | | | 81 |
| Iterate | | | | | 41 |
| LN | | | | | 60,61 |
| LOAD LAYER | | | | | 27 |
| LOCAL | | | | | 28,29,34,57,58 |
| Localised error-trapping | | | | | 33,34 |
| Logical expressions | | | | | |
| Logical operators | | | | | 39 |
| LoRes | 81 | 84 | 02 | -3, 04 | 16,14 |

M

| | | | | | |
|-------------------------|---|---|--|--|-------------|
| Mathematical Functions | + | + | | | 60,61,62,63 |
| Mathematical operations | | | | | 37 |
| MID\$ | | | | | 58 |
| MOD | | | | | 37,47 |
| MOD files | | | | | 54 |
| MOVIE | | | | | 42 |
| Moving Sprites | | | | | 134, 35 |
| MP3 files | | | | | 56 |
| MSB | | | | | 02 |

N

| | | | | | |
|-------------------|---|---|---|---|------------|
| Natural logarithm | + | + | | | 61 |
| Natural scale | | | | | 46 |
| NEW | | | | | 50,14 |
| NextHEC | | | | | 40 |
| NEXT | | | | | 2,27,23,24 |
| NEXT # | | | + | + | 126 |
| Next Register | + | + | + | + | 24 |
| NextBASIC Menu | | | | | 117 |
| NextZXS | | | | | 68 |
| NMI Menu | | | | | 34 |
| NOT | | | | | 18, 17, 13 |
| Note duration | | | | | 149, 50 |
| Note volume | | | | | 5 |
| null string | | | | | 4 |
| numeric arrays | | | | | 68 |

O

| | | | | | |
|-----------------------|----|---|---|---|---------------|
| octave command | ++ | + | + | + | 49 |
| ON | | | | | 6,19,20,36 |
| ON ERROR | | | | | 39 |
| OR | | | | | 38,46,7,72,73 |
| Order of calculations | | | | | 52 |
| OUT | | | | | 80,10 |
| OVER | | | | | 86,0 |

P

| | | | | | |
|----------------|--------|--------|--|--|---------|
| PALETTE | 03,107 | 08, 79 | | | 1,2,43 |
| PALETTE CLEAR | | | | | 09 |
| PALETTE DIM | | | | | 108, 36 |
| PALETTE FORMAT | | | | | 08 |

| | | | | | |
|-------------------------|--|--|--|--|-------------------------------|
| PRINTUS | | | | | 15, 11 |
| PAPER | | | | | 75,95, 00, 105, 08, 09, 5, 30 |
| parameters by reference | | | | | 72 |
| PAUSE | | | | | 21, 23, 24 |
| PAUSE STOP | | | | | 42 |
| PEEK | | | | | 58,78, 7, 18,80, 22, 25, 28 |
| PI | | | | | 6 |
| peek | | | | | 96,98 |
| PI | | | | | 6 |
| PIA | | | | | 45, 47, 48, 13, 15 |
| PLOT | | | | | 88, 1, 4, 5, 6, 20 |

PLC INVERSE

PL INVERSE

| | | | | | |
|------------------------|--|--|--|--|---------------------------------|
| POINT | | | | | 84,85,86,87,91, 9, 16, 17 |
| POINT TO | | | | | 14,117 |
| POKE | | | | | 78,77,78,79,80,92,93,97, 22,128 |
| prodimensioned | | | | | 67 |
| PRINT | | | | | 5,20,89,21, 23,47, 45,76,8 |
| | | | | | 82,84,86,84,3, 76, 5 |
| PRN | | | | | 20,89 |
| PRINT AT | | | | | 83,85,88 |
| PRINT com | | | | | 82,85,85,90 |
| PRINT POINT | | | | | 85 |
| PRINT separator | | | | | 82 |
| priority colours | | | | | 1,2 |
| PRIVATE | | | | | 29 |
| PRIVATE CLEAR | | | | | 28 |
| PROC | | | | | 29,32,3, 32,33,42 |
| PROCedure | | | | | 2 |
| Procedures | | | | | 29,30,3 |
| Procrustean assignment | | | | | 50,69 |
| PSDs | | | | | 44, 49 |

R

| | | | | | |
|----------------------|--|--|--|--|--------------------|
| RIGSB2 | | | | | 95, 11 |
| RIGSB3 | | | | | 96 |
| adians | | | | | 63 |
| RANDOM | | | | | 25, 26 |
| RANV | | | | | 65 |
| random | | | | | 84 |
| RANDOMIZE | | | | | 84,65,66 |
| raster line | | | | | 4 |
| REAL | | | | | 33,35,35,42,70, 30 |
| recursion | | | | | 72 |
| REF | | | | | 32,58 |
| REG | | | | | 58,80, 23, 28, 4 |
| Relational operators | | | | | 19 |
| relative source | | | | | 76, 38 |
| REPEAT | | | | | 23,14,21, 26 |
| REPEAT UNTIL | | | | | 23,24,25,26 |
| resolution | | | | | 96 |
| RESTORE | | | | | 55,38 |

| | |
|----------------------------------|----------------------------|
| RELUKN | 19,24,27,29 |
| RGB | 95 |
| RIGHTS | 58 |
| RND | 55,64,65,66 |
| RND % | 64,65 |
| ROM | 89 |
| row by column coordinate | 97 |
| RTC | 122,125 |
| RUN | 19,35,36,47,89,114,132,134 |
| S | |
| SAVE | 130 |
| SAVE _ BANK | 130 |
| Scientific notation | 40 |
| screen memory | 97 |
| SCREENS | 88 |
| Scrolling | 89,140 |
| Scroll-prompt inhibitor | 92 |
| SD | 126 |
| SDH files | 165 |
| SELECT CASE | 19 |
| select operator ? | 16,19 |
| semisigs | 146 |
| SGN | 55 |
| SGN {..} | 39,47,49 |
| SHIFT | 93 |
| SID files | 154 |
| Signed | 45,46,47,48 |
| signed expression | 48 |
| signed integer expressions | 47 |
| SIN | 60,62,63 |
| sine | 62 |
| single-dimension character array | 70 |
| SPACE | 90 |
| spatial resolution | 96 |
| SPRITE | 108,113,130,131,136,137 |
| SPRITE AT | 139 |
| SPRITE BANK | 130,131 |
| SPRITE BORDER | 130,132 |
| SPRITE CLEAR | 130 |
| SPRITE CONTINUE | 137,139 |
| SPRITE DIM | 132 |
| Sprite functions | 139 |
| Sprite Layer | 81 |
| SPRITE MOVE | 137 |
| SPRITE MOVE INT | 137 |
| SPRITE OVER | 139 |
| SPRITE PALETTE | 135 |
| SPRITE PALETTE BANK | 135 |
| SPRITE PAUSE | 139 |
| sprite ports | 128 |
| SPRITE PRINT | 130,131 |
| SPRITE (function) | 139 |
| SPRITE STOP | 137 |
| Sprite System | 101,128 |
| SQR | 56,59 |
| STEP | 22,23,26 |
| Stereo control | 153 |
| stripping | 118 |
| STOP | 90 |
| STR | 53 |
| Streams | 123 |
| String | 53 |
| String functions | 63 |
| String multiplication | 50 |
| String slicing | 49 |
| subexpression | 48 |
| subscript | 67 |
| subscripted variables | 67,68 |
| subscripts | 68 |
| substring | 58 |
| SYMBOL SHIFT | 90 |
| System Variables | 93,122 |
| T | |
| TAB | 82,86,89,92 |
| tabulating character | 88 |
| TAN | 60,62,63 |
| tangent | 62 |
| Tempo | 152 |
| Text windows | 82 |
| The Copper | 140,141,142,143 |
| THEN | 16,17 |
| tile | 80 |
| TILE | 113,119 |
| TILE AT | 119 |
| TILE BANK | 119 |
| TILE DIM | 119 |
| tile offset | 120 |
| TILE w,h AT x,y | 119 |
| TILE w,h AT x,y TO k2,y2 | 120 |
| TILE w,h TO k2,y2 | 119,120 |
| tilemap | 119 |
| tiles | 119,120 |
| TIME | 122,124 |
| TIMES | 123,124 |
| Times | 81 |
| TL\$ | 58 |
| TO | 21,22,32,49 |
| Tokenisation | 51 |
| trailing modifiers | 51 |
| transparency | 130 |
| transparency colour index | 109 |
| transparency colour mask | 109 |

| | | | |
|-----------------------------|-------------|--------------------|--------------|
| True | 71,72 | VAL | b1,b2,b3,124 |
| two-dimensional array | 68 | VALS | 51,55 |
| TZX files | 156 | Variable names | 39 |
| U | | Variable not found | 92 |
| UART | 155 | vertical size | 96 |
| UDG | 49,75,128 | visibility | 136 |
| ULA | 103 | visibility flag | 136 |
| Unary ! | 48 | Volume effects | 151 |
| unary not | 46 | W | |
| Unary/Bitwise NOT (!) | 39 | WAIT | 140,141 |
| unified write | 137,138 | WAV | 154 |
| Unsigned | 45,46,47,48 | WHILE | 25 |
| unsigned expression | 48 | WRITE | 142 |
| user-defined graphic | 76 | X | |
| user-defined graphics | 75 | XOR | 45 |
| Using Expressions for INPUT | 91 | | |
| USR | 59,75,76 | | |

ZX Spectrum Next Home Computer

Technical Specifications

Issue 2 Model

- Xilinx Spartan-6™ SLX16 FPGA (XC6SLX16) implementing
 - ▶ CPU: Z80N CPU with extended instruction set @ 3.5/7/14/28 MHz
 - ▶ All standard ZX Spectrum and Timex video-modes with the addition of Layer 2, Layer 3 and LoRes video with 9 bit colour and hardware scrolling
 - ▶ TurboSound compatibility (3 PSGs) - 3 x AY-3-8912 or YM-2149 compatible PSG audio chips with stereo output
 - ▶ Covox™/Soundrive™/SpecDrum™ compatible digital audio
 - ▶ Selectable DMA controller (Z80DMA and zxnDMA)
 - ▶ Amiga™ like Copper hardware
 - ▶ Hardware Sprite Engine
 - ▶ EnhancedULA extending legacy Spectrum and Timex modes to 256 colours out of a 512 colour palette
 - ▶ Multiface compatible NMI handling hardware
 - ▶ Two programmable UARTs
 - ▶ Programmable CTC chip (8 channels)
 - ▶ I²C bus
 - ▶ SPI bus
 - ▶ PS/2 keyboard and mouse controller
 - ▶ Two joystick controllers compatible with Kempston, Sinclair and Cursor standards
 - ▶ divMMC interface with an external SD card slot (as well as additional possibility for a secondary internal micro SD card slot, only via expert soldering at user's own risk)
- Memory: 1MB SRAM (expandable to 2MB SRAM)
- 58 key laptop-style low profile tactile matrix keyboard
- Video out ports: RGB / VGA Analog and HDMI-compatible Digital Video Port
- Stereo Audio Out port
- Two multipurpose controller and I/O ports for joystick and serial communications
- Tape support, with joint Mic and Ear ports
- Original external bus expansion port
- Internal accelerator expansion port (for optional Raspberry Pi Zero¹ accelerator)
- Optional I²C RTC (Real Time Clock) device (DS-1307)
- Optional Wi-Fi module, with a full TCP/IP stack (ESP8266).
- On board GPIO for user expansion
- Multicore capability

¹ A Raspberry™ Pi Zero connects into the accelerator expansion port, which gives you one micro-USB port and an additional mini-HDMI output. The Raspberry Pi Zero comes with a 1 GHz CPU, a GPU and 512 MB of RAM, and brings yet more possibilities to your ZX Spectrum Next, such as supporting a second display, additional sound playback and processing and even more advanced graphics processing power.

Issue 4 Model

- Xilinx Artix-7™ A15T FPGA (XC7A15T) implementing:
 - ▶ CPU Z80N CPU with extended instruction set @ 3.5/7/14/28 MHz
 - ▶ All standard ZX Spectrum and Timex video-modes with the addition of Layer 2, Layer 3 and LoRes video with 9 bit colour and hardware scrolling
 - ▶ TurboSound compatibility (3 PSGs) – 3 x AY-3-8912 or YM-2149 compatible PSG audio chips with stereo output
 - ▶ Covox™/Soundrive™/SpecDrum™ compatible digital audio
 - ▶ Selectable DMA controller (Z80DMA and zxDMA)
 - ▶ Amiga™ like Copper hardware
 - ▶ Hardware Sprite Engine
 - ▶ EnhancedULA extending legacy Spectrum and Timex modes to 256 colours out of a 512 colour palette
 - ▶ Multiface compatible NMI handling hardware
 - ▶ Two programmable UARTs
 - ▶ Programmable CTC chip (8 channels)
 - ▶ I²C bus
 - ▶ SPI bus
 - ▶ PS/2 keyboard and mouse controller
 - ▶ Two joystick controllers compatible with Kempston, Sinclair and Cursor standards
 - ▶ divMMC interface with an external SD card slot (as well as additional possibility for a secondary internal micro SD card slot, only via expert soldering at user's own risk)
- Memory 2MB SRAM standard
- 58 key laptop-style low profile tactile matrix keyboard
- Video out ports: RGB / VGA Analog and HDMI-compatible Digital Video Port
- Stereo Audio Out port
- Two multipurpose controller and I/O ports for joystick and serial communications
- Tape support, with joint Mic and Ear ports
- Original external bus expansion port
- I²C RTC (Real Time Clock) device (DS-1307)
- Wi-Fi module, with a full TCP/IP stack (ESP8266).
- Internal accelerator expansion port (for optional Raspberry Pi Zero accelerator)
- On board GPIO for user expansion
- Multicore capability